Proceedings

LOGIC PROGRAMMING WORKSHOP'83

Praia da Falésia, Algarve / PORTUGAL

Núcleo de Inteligência Artificial
UNIVERSIDADE NOVA DE LISBOA

Proceedings

LOGIC PROGRAMMING WORKSHOP '83

Praia da Falesia, Algarve / PORTUGAL

26 Jun - 1 July, 1983

Edited by:

Luís Moniz Pereira
António Porto
Luís Monteiro
Miguel Filgueiras

Sponsored by:

Associação Portuguesa para a Inteligência Artificial
Direcção-Geral do Ensino Superior
Junta Nacional de Investigação Científica e Tecnológica
Instituto Nacional de Investigação Científica

Núcleo de Inteligência Artificial

UNIVERSIDADE NOVA DE LISBOA

F O R E W O R D

These are the papers presented at the LOGIC PROGRAMMING WORKSHOP'83, which took place at Hotel Alfa-Mar, Praia da Falesia, Albufeira, Algarve, in Portugal, from June 26 to July 1, 1983.

The workshop was organized by the Nucleo de Inteligencia Artificial, of Departamento de Informatica, Faculdade de Ciencias e Tecnologia, Universidade Nova de Lisboa.

We thank everyone who helped us in organizing this event, and all the participants for their contribution.

We further thank, for their financial support :

Direccao Geral do Ensino Superior
Instituto Nacional de Investigacao Cientifica
Junta Nacional de Investigacao Cientifica e Tecnologica

Thanks are also due to the Servicos Graficos da Universidade Nova de Lisboa.

We expect that a refereed selection of these papers will be soon published in book form.

To obtain a copy of the proceedings delivered by air mail send 1,700 Escudos or equivalent, by personal check or otherwise, to

Logic Programming Workshop'83
Nucleo de Inteligencia Artificial
Universidade Nova de Lisboa
2825 Quinta da Torre
Portugal

The program chairman

Luis Moniz Pereira

C O N T E N T S

Some Reflexions on Implementation Issues of PROLOG

M.Bruynooghe

Departement Computerwetenschappen
Katholieke Universiteit Leuven

Celestijnenlaan, 200 A
B 3030 HEVERLEE

INTRODUCTION

Current interest in PROLOG is high. This papers aims at opening a discussion on implementation related issues which, in our opinion, can have a great impact on the acceptance of PROLOG as a valuable programming language. Our focus is on issues concerning users of todays PROLOG, not on implementation issues in current research on logic programming (parallellism, intelligent backtracking, special architecture, control).

The issues are:

- the bad influence of cut on programming style but its necessity, in current implementations, to obtain efficient (time and space) execution of programs.
- Can/will everyone develop good (efficient) PROLOG programs or is this an art for a small club of skillful experts.
- an observed desire for standardisation.

1 The influence of an implementation on programming style.

The PROLOG community has gone a long way from the first PROLOG interpreters to current compilers promising to allow efficient execution without running out of space, even for infinite queries if they are determinate.

In the first interpreters, we could distinguish two major work areas:

1 a dictionary of clauses, spoiled but slowly because "retract" does not free the space.

2 an environmentstack ("trail" included or separate) which grows until backtracking liberates it.

This resulted in "dirty" programstyle, exemplified by the following:

When you are affraid of running out of core, then:

- assert your useful results.
- fail and backtrack.
- restart, picking up your useful results and retract their assertion.

It has been learned to separate the global/copystack from the environmentstack, to keep the environmentstack small by exploiting determinism and to apply garbage collection on the copystack. Also compilation techniques have been developped to obtain more efficiency.

Does the combination of all these features provide the paradise for the purists among logic programmers willing to apply such an advanced programming technique as the usage of abstract data types ?

Let us look at a simple example:

A procedure Partition(x, l, l1, l2) which separates a list l into a list l1 of elements less than or equal to x, and a list l2 of elements greater than x.

The datastructure can be implemented as follows:

Empty(l) : a test to see whether a list l is empty, also to initialize an empty list.

A possible realisation is Empty(Nil) <- (another one could be with difference lists: Empty(d(x,x)) <-)

Select(l, x, tail): a list l is separated into its first element x and its remainder (tail) (Fails for an empty list)

Realisation: Select(x.l, x, l) <-

Construct(x, tail, l): a list l is constructed with first element x and remainder tail

Realisation: Construct(x, l, x, l) <-

Now Partition can be defined:

```
Partition(x, l, l1, l2) <- Empty(l), Empty(l1), Empty(l2)

Partition(x, l, l1, l2) <- Select(l, e, l'), e <= x,
                           Construct(e, l1', l1), Partition(x, l', l1', l2)

Partition(x, l, l1, l2) <- Select(l, e, l'), e > x,
                          Construct(e, l2', l2), Partition(x, l', l1, l2')
```

To my knowledge, the best of all existing PROLOG systems cannot prevent that a heavy price is paid for this programming style:

- Efficiency: the recursive calls contain 4 calls instead of 2, the number of logical inferences required to obtain a solution is doubled.

- Space: current implementations are unable to recognize the determinism of the above program. Half of the calls will be considered as nondeterministic, backtrackpoints will be created and will stay on the environmentstack. Completion of a partition call will not free the environmentstack. As a consequence, references to the global/copystack are not removed and the potential for garbage collection is severely reduced.

Preprocessing the calls to Empty, Select and Construct can improve the situation. A technique as "partial evaluation" seems promising to automate this. Doing the partial evaluation by hand, we arrive at:

```
Partition(x, Nil, Nil, Nil) <-
Partition(x, e.l', e.l1', l2) <- e <= x, Partition(x, l', l1', l2)
Partition(x, e.l', l1, e.l2') <- e > x, Partition(x, l', l1, l2')
```

This solves the efficiency problem but not the space problem (to recognize the determinism of the base case (empty list), indexing on the second argument is necessary)

To recognize determinism, a cut in the second clause is needed. At the same time, an experienced programmer will drop the condition in the third clause. We obtain:

```
Partition(x, Nil, Nil, Nil)<-
Partition(x, e.l', e.l1', l2') <- e <= x, /, Partition(x, l', l1', l2)
Partition(x, e.l', l1, e.l2') <- Partition(x, l', l1, l2')
```

This cut, an ugly feature to purists and beginners, changes the whole nature of the program. Now, the program Partition(x, l, l1, l2) cannot be used to obtain all possible merges l of l1 and l2. This restriction on the use of Partition is not declared. As in most cases, it is the purpose of the cut to inform the execution mechanism about determinism, about the opportunity to optimise the execution of the program. At the same time, the possible use of the program is severely restricted. It is a sad fact that this guidance and its accompagnying restriction are not given on a more elegant and more explicit way. It is the obscurity of the cut which restricts the use of PROLOG, beyond toy examples, to skillful expert programmers having a good understanding of the underlying implementation.

Actually, we can state two questions about the above program, (I am affraid that studying the manual of a particular implementation will not answer them):

1 Is a cut needed in the first clause to recognize the determinism of the base case (Nil)? Not recognizing the base case as deterministic has a dramatic effect on memory usage, frames are locked on the stack!

2 Will tail recussion optimisation be obtained when the second clause is selected ? It cannot be applied at the time of unification with the heading because the call is nondeterministic (the third clause provides an alternative), it becomes deterministic only after execution of the cut; at that moment, the opportunity exists to collapse two stack frames into one.

2 Applicability of PROLOG

The application of PROLOG is rather limited. It is only used by skillfull experts, mainly in the field of artificial intelligence. Can it be applied to more conventional problem areas, where Fortran, Cobol, Basic or Pascal are used, i.e business applications ? Recently, we conducted a few experiments. Our tools: a slow PROLOG interpreter written in Pascal (about 400 logical inferences per second on a VAX 750) , the vendors Basic and Pascal compiler.

Experiment 1

A parser was developed by rather unexperienced programmer using a compiler generator system semantic actions hand-written. The result: 2276 lines of Pascal. A parser which is roughly functionnally equivalent was written in PROLOG by an expert PROLOG programmer: 246 lines of PROLOG (factor 9). Execution time (parsing the same file): 52 s (seconds) with Pascal, 296 s with PROLOG (factor 5 - 6).

Experiment 2

A program with complex data structures. An unexperienced Pascal programmer: 2371 lines of code, a skillful PROLOG programmer: 136 lines of PROLOG (factor 17). Both programs have roughly the same functional equivalence. Execution times: Pascal: 43 s, PROLOG: 119 s. (factor 3).

Experiment 3

A complex retrieval task involving 4 files on the vendors file system (RMS). A Basic program of 170 lines required an execution time of 12.5 sec. A PROLOG program with exactly the same functionality required 70 lines (factor 12.5) and an execution time of 191 s. (factor .17). (Due to the experimental nature of the interface PROLOG-RMS, each call to the file system required a lot of additional logical inferences, also the program did more file accesses.). A, for PROLOG, simple optimisation (bringing a small part of 2 files in core) reduced the execution time to 47 s. (factor 4).

Taking into account the slowness of the interpreter, an

improvement of factor 5 to 10 (2000 - 4000 LIPS) seems not difficult to obtain, using compilation techniques, further improvements are possible,this suggests that PROLOG becomes a competitive language to be used on a large scale when:

- complex data structures are to be manipulated (making use of the full power of unification)
- a substancial amount of time is spend on file accesses.

However, this requires:

- A robust interpreter, not running out of space while executing large programs and sufficiently efficient.

- Easily extendible set of evaluable predicates, allowing to develop specific predicates for specific applications. e.g.:

  * connection with a particular file system or database
  * screen management functions
  * allowing to implement components with insufficient efficiency at a lower level.

- An environment allowing the development of good PROLOG programs by mediocre programmers, this probably requires:

  * A cut free variant of PROLOG, making less an art of the writing of space efficient programs (see reflection 1)

  * An automatic optimiser based on partial evaluation techniques (see reflection 1)

  * More compile time verification (types, restrictions on the usage of procedures), (Preferably incremental and integrated in a syntax oriented editor)

- A lot of programming is involved with side effects which should be in a particular order. e.g. interaction with a terminal, producing a report. We need a well choosen set of metalevel predicates to control such side-effects. Some possibilites:

  * For_each (cond, action), e.g.:
    For_each (employee(x), compute_salary(x))

    (Definition in Prolog :
     For_each(cond,action) <- cond, action, fail
     For_each(cond,action) <-

  * Repeat_until_exit(command), e.g.:
    Repeat_until_exit(Read_and_process(x))
    The execution backtracks, reading commands until a special "exit" call is executed and the infinite backtracking is destroyed.

  Although Prolog allows every expert to implement his own set of such metapredicates, some standardisation is desirable and integration of

them in compiletime verification tools is needed.

3 Portability of Prolog programs

Currently, Prolog is not only a subject for research, but becomes accepted as a suitable language for implementing diverse applications (e.g. its role in the Japanese Fifth Generation Project and in the European ESPRIT project). This creates the problem of exchanging programs, of portability of programs. Although Edinburgh's DEC10 Prolog tends to be a de facto standard, different implementations exist and many more are likely to appear. Porting Prolog programs from one Prolog system to another is problematic due to :

- Differences in syntax. As far as the syntax is syntactical sugar for Horn clauses, automatic conversion seems not difficult. However, conversion to Horn clauses poses a problem when the syntax allows for alternatives (e.g. (P;Q)) inside a clause, in cases where such an alternative contains different calls and one of them is a cut (scope of the cut).
- Differences in evaluable predicates. Although the core of Prolog is extremely simple, manuals are becoming wieldy, they look like christmas trees, full of evaluable predicates. Some have to do with high level control of the system, but others are extensively used inside programs.

The observable desire to use Prolog as a programming language for large projects creates a need for standardisation of syntax and evaluable predicates. Is it possible to define a minimal set of evaluable predicates and to define all extensions as Prolog procedures? For reasons of efficiency, an implementor can provide these extensions at a lower level. Taking Edinburgh's Prolog as the de facto standard is probably not optimal, some reflection on the choice seems preferable.

A Prological Definition of HASL a Purely Functional Language with Unification Based Conditional Binding Expressions

Harvey Abramson

Department of Computer Science
University of British Columbia
Vancouver, B.C. Canada

ABSTRACT

We present a definition in Prolog of a new purely functional (applicative) language HASL (*H. A*bramson's *S*tatic *L*anguage). HASL is a descendant of Turner's SASL and differs from the latter in several significant points: it includes Abramson's unification based conditional binding constructs; it restricts each clause in a definition of a HASL function to have the same arity, thereby complicating somewhat the compilation of clauses to combinators, but simplifying considerably the HASL reduction machine; and it includes the single element domain {fail} as a component of the domain of HASL data structures. It is intended to use HASL to express the functional dependencies in a translator writing system based on denotational semantics, and to study the feasibility of using HASL as a functional sublanguage of Prolog or some other logic programming language. Regarding this latter application we suggest that since a reduction mechanism exists for HASL, it may be easier to combine it with a logic programming language than it was for Robinson and Siebert to combine LISP and LOGIC into LOGLISP: in that case a fairly complex mechanism had to be invented to reduce uninterpreted LOGIC terms to LISP values.

The definition is divided into four parts. The first part defines the lexical structure of the language by means of a simple Definite Clause Grammar which relates character strings to "token" strings. The second part defines the syntactic structure of the language by means of a more complex Definite Clause Grammar and relates token strings to a parse tree. The third part is semantic in nature and translates the parse tree definitions and expressions to a variable-free string of combinators and global names. The fourth part of the definition consists of a set of Prolog predicates which specifies how strings of combinators and global names are reduced to "values", ie., integers, truth values, characters, lists, functions, fail, and has an operational flavour: one can think of this fourth part as the definition of a normal order reduction machine.

April 24, 1983

A Prological Definition of HASL a Purely Functional Language with Unification Based Conditional Binding Expressions

Harvey Abramson

Department of Computer Science
University of British Columbia
Vancouver, B.C. Canada

1. Introduction

In this paper we shall use Definite Clause Grammars (DCGs) and Prolog to present a definition of HASL, a purely functional language incorporating the unification based conditional expressions introduced in [Abramson,82a].

Metamorphosis grammars were introduced in [Colmerauer,78] and were shown to be effective in the writing of a compiler for a simple programming language. Definite Clause Grammars, a special case of metamorphosis grammars were introduced in [Pereira&Warren,80] and shown to be effective in "compiling", ie, translating a subset of natural language into first order logic. Metamorphosis grammars (M-grammars) have been used to describe several languages, namely ASPLE, Prolog, and a substantial subset of Algol-68 [Moss,81], [Moss,79]; see also [Moss,82] for the use of Prolog and grammars as tools in language definition. Although neither M-grammars nor DCGs were mentioned in [Warren,77], that paper is of interest in the use of Prolog as a compiler writing tool. The use of DCGs and Prolog for the implementation of SASL [Turner,76,79,81], a purely applicative language, was reported in [Abramson,82b].

The language HASL which we shall define below arose out of the Prolog implementation of SASL. One reason for defining the new language was to incorporate unification based conditional binding expressions; another was to simplify and clean up the combinator reduction machine introduced by Turner to evaluate SASL expressions; a third was to provide a possible functional sublanguage for Prolog; and a fourth was to provide a functional notation for a denotational semantics based translator writing system akin to Mosses' Semantics Implementation System [Mosses,79] but to be tied to DCGs. Although we do not present a purely logical definition of HASL, we feel that the departures from Horn clause logic in the definition presented below (the use of the cut for control; negation as failure; and extension of HASL's database of globally defined functions) are not significant enough to mar the formality of the definition or its comprehensibility. The definition can be used as a specification of HASL, as an interpretive implementation of HASL, and as a guide to a more efficient implementation of HASL in some system programming language.

In section 2 we shall informally and briefly describe HASL. Section 3 contains a description of the general definition strategy: HASL expressions are compiled to variable-free strings of combinators, global names, and uses of the two primitive operations of function application (—>) and pair construction(:, like LISP's CONS). Following this are sections devoted to: the DCG for lexical analysis; the DCG and associated predicates which perform syntactic analysis and parse tree formation; the translation to combinators; and the HASL reduction machine. A final section will suggest some further work which we intend to pursue.

2. HASL - Informally and Briefly

HASL is descended from Turner's SASL (see [Turner,76,79,81]) and obviously owes much to it. We have chosen to designate this language HASL not to suggest that what we present is totally original, but that there are enough departures from SASL to warrant a new designation.

A HASL program is an expression such as

```
[1,2,3] ++ [4,5,6]
```

with value

```
[1,2,3,4,5,6]
```

or an expression with a list of equational definitions qualifying the expression:

```
f x
where
x= hd y
hd (a:x) = a,
y = 3:y,
f 0 = 1,
f x = x * f(x-1)
```

with value 6.

We note in this list of definitions that

[1] A function such as *f* may be defined by a list of clauses. The order of the clauses is important: in applying *f* to an argument the first clause will be "tried", then the second, etc.

[2] In the definition of *hd* the argument must be a constructed pair, specified by *(a:x)* where : is the HASL pair constructor. Structure specifications may involve arbitrary list structures of identifiers and constants.

[3] HASL makes use of lazy evaluation ([Henderson & Morris,76]) so that infinite lists such as *y* may be defined, and elements of such lists may be accessed, as in *hd y* without running into difficulties.

A list may be written as

```
[1,2,3]
```

which is syntactic sugaring for

```
1: 2: 3: []
```

where [] denotes the empty list and the notation 'string' is a sugaring for the list of character denotations:

```
%s: %t: %r: %i: %n: %g: []
```

There are functions such as *number, logical, char* and *function* which may be used to check the types of HASL data objects:

```
number 12 = true
logical 5 = false
char %%   = true
function hd = true
```

are all HASL expressions which have the value true.

Functions may be added to HASL's global environment as follows:

```
def
string [] = true,
string (a:x) = char a & string x,
string x = false,
cons a b = a:b
;
```

Each clause defining a HASL function *f* must have the same number of arguments or arity. Thus above, each clause in the definition of *string* has arity one. In the second clause for *string* however, a single structured argument is designated. Although HASL functions may be written as if they had several arguments, such as *cons* above, HASL functions are all considered to have in fact single arguments. The single argument is a HASL data object which may be a character, a truth value, an integer, fail, a list of HASL objects, or a function of HASL objects to HASL objects. The value of such a function may be any HASL object - including a function. Thus the value of

```
cons %a
```

is the HASL function which puts *%a* in front of lists.

The HASL object *fail* is the result of, for example, applying *hd* to a number:

```
hd 5 = fail
```

The object *fail* is not a SASL object and is one of our departures from that language.

Another departure is in the introduction of the restricted unification based conditional binding constructs {- and -} of [Abramson,1982].

```
formals {- exp1 => exp2 ; exp3
```

The meaning of this is that if *exp1* can be unified to the list of *formals,* then the value of this expression is the value of *exp2* qualified by the bindings induced by the match; otherwise, it is the value of *exp3.* This may be expressed somewhat inefficiently using the HASL conditional expression *(a -> b ; c):*

```
(fail = f exp1
 where f formals = exp2) ->
exp3 ;
(f exp1
 where f formals = exp2)
```

Thus the unification expression may be regarded as the definition and application of an anonymous function.

The unification expression is in fact the basis of the compilation of HASL clausal definitions into a single function. If *member* is defined by the following clauses:

```
def
member a [] = false,
member a (a:x) = true,
member a (b:x) = member a x
;
```

then the HASL specification and interpreter treats this as:

```
member x1 x2 =
a [] {- x1 x2 => false;
a (a:x) {- x1 x2 => true;
a (b:x) {- x1 x2 => member a x;
fail
```

3. The Top Level of the HASL Specification.

A HASL expression denotes a value. We may express this by the notation

hasl(Expression,Value).

This relation requires some refinement, however. The expression is written as some sequence of characters, including spaces, carriage returns, etc., and the characters must be grouped into a sequence of meaningful HASL "tokens". These tokens must then be grouped into meaningful syntactic units determined by the syntax of HASL expressions. These two relations, the lexical and syntactic, are expressed by means of two DCGs: one DCG defines the relation between a sequence of characters and a sequence of HASL tokens; a second DCG defines the relation between a sequence of HASL tokens and a representation of the syntactic structure of a HASL expression as a tree.

Further, the expression of the relation between the tree and the value denoted by the original sequence of characters requires refinement. The tree represents the abstract syntax of the HASL expression. A semantic relation holds between this tree and a sequence of combinators, global names, function application operators (--->) and pair construction operators (:). This relation therefore defines a translation from a syntactically sweet string of symbols (HASL) to a mathematically equivalent - but rather unreadable - sequence of symbols suitable for mechanical evaluation or reduction. The reduction relation (=>>) specifies how such a sequence of symbols is related to another sequence of symbols which is the head normal form of the first. A final relation (>>>) between head normal form and HASL values (normal form) completes the specification of the relation *hasl:*

```
hasl(Expression,Value) :-
lexical(Expression,Tokens),
syntactic(Tokens,Tree),
semantic(Tree,Combinators),
Combinators =>> HeadNormal,
HeadNormal >>> Value.
```

The lexical relation may be specified in terms of a relation *lexemes* (see next section):

```
lexical(Expression,Tokens) :- lexemes(Tokens,Expression,[]).
```

and the syntactic relation may be specified in terms of a relation *expression* (see Section 5):

```
syntactic(Tokens,Tree) :- expression(Tree,Tokens,[]).
```

The two relations *lexemes* and *expression* are defined below by definite clause grammars.

4. The Lexical Specification of HASL.

This relation requires little comment. A sequence of characters such as

"def fac 0 = 1, fac x = x * fac(x-1);"

is grouped into the following string of tokens:

```
[def,id(fac),constant(num(0)),op(3,cEQ),constant(num(1)),comma,id(fac),id(x),
op(3,cEQ),id(x),op(5,cMULT),id(fac),lparen,id(x),op(4,cSUB),constant(num(1)),
rparen,semicolon]
```

Identifiers such as *fac* are represented by *id(fac),* constants such as 0 are represented by *constant(num(0)).* Some reserved words and punctuation are represented by atoms such as *def* and *comma.*

A sequence of definite clause rules such as

```
tIDENT(id(Id)) -> [id(Id)].
```

defines the function symbols which are the terminals for syntactic analysis.

The complete Prolog specification of HASL is at the end of this report following the References.

5. The Syntactic Specification of HASL.

As mentioned above, the *syntactic* relation is between token strings and parse trees which represent the abstract syntax of HASL expressions.

The leaves of a parse tree may be identifiers such as *id(fac)*, constants such as *logical(true)*, *num(123)*, *char(C)*, *fail*, or they may be the names of certain known HASL combinators such as *cADD* for addition, or *cMATCH* used in unification, etc. These names follow the convention of a lower case *c* followed by some other letters (usually upper case), digits or underline characters.

There are several kinds of branch nodes. A branch may be labeled by the HASL function application arrow (—>) or by the HASL pair construction colon (:). The arrow associates to the left, the colon to the right. Thus the linear parse tree representation of *a+ 1* is:

```
cADD —> id(a) —> num(1)
```

and that for *hd 'abc'* is:

```
cHD —> %a : %b : %c : []
```

Another kind of branch node is labeled with the functor *where* and has one subtree which is an expression and another which is the subtree for a list of definitions qualifying the expression:

```
where(Exp,Defs)
```

Global definitions are subtrees of a tree where the root is labeled with the functor *global*

```
global(Defs)
```

To each definition there is a branch node labeled with the functor *def* and with three subtrees: the name of the identifier being defined; the arity associated with the name being defined; and, the expression or list of clauses to be associated with the name. For a name with arity 0 such as in:

```
def b = a + 1;
```

the definition node looks like:

```
def(id(b),0,cADD—>id(a)—>num(1))
```

When a function is being defined, the arity is at least one, and the third argument is a list of clauses, each of the form:

```
func(Fseq,Exp)
```

where *Fseq* is a list of arguments of length *arity* for the function being defined, and *Exp* is the expression associated with that clause. Thus, the definition of *member* in Section 2 is represented in a parse tree as:

```
def(id(member),2,
[func([id(a)|flist(id(b):id(x))],id(member)—>id(a)—>id(x)),
func([id(a)|flist(id(a):id(x))],logical(true))|
func([id(a)|const(nil)],logical(false))])
```

The functor *flist* is used to label a branch of a tree in which a list structured argument to a function is specified. The context sensitive restriction that each clause defining a function have the same arity is specified by the predicate *mergedef* which merges separate clauses for a function into

one node of the above description. (See the next two sections for further discussion of this restriction.)

One other point to note is that a list of definitions of arity 0 such as

[x,y,z] = x

is represented as a list of definition nodes:

[def(id(x),0,cHD—>id(x)),
def(id(y),0,cHD—>(cTL—>id(x))|
def(id(z),0,cTL—>(cTL—>id(x))]

This is specified by the predicate *expandef.*

The last remaining kind of branch node is that for a unification based conditional binding expression.

(a:x){-y=>x;fail
y-}(a:x)=>x;fail

would both be represented as:

unify(flist(id(a):id(x)),id(y),id(x),fail)

The DCG specifying the syntax of HASL is fairly straightforward. There is some slight intricacy in the specification of the grammar rules for expressions involving the HASL operators: operator precedence techniques are used to build the appropriate subtrees.

The function symbols beginning with a lower case *t* are the terminals for this grammar and specify HASL tokens as defined by the lexical DCG.

6. The Semantic Specification of HASL.

The *semantic* relation is one which holds between parse trees as specified in the previous section, and certain strings of combinators, constants, global names, and the primitive HASL operations of function application (—>) and pair construction (:). These strings may in fact be regarded as modified parse trees in which the *global, where, def, func, flist* and *unify* nodes have been eliminated and replaced by variable-free subtrees. The elimination of these nodes depends on a discovery of the logician Schoenfinkel: that variables, although convenient, are not necessary.

Schoenfinkel's discovery that variables can be dispensed with relies on a sort of cancellation related to extensionality. If in HASL we defined

successor x = plus 1 x
plus a b = a + b

then we could say that

successor = plus 1

for both sides, when applied to the same argument, are always equal.

Schoenfinkel related a variable, an expression which may contain that variable, and an expression from which that variable had been abstracted (removed) with the aid of the following combinators:

cS x y z = x z (y z)
cK x y = x
cI x = x

The specification of the abstraction or removal of a variable is given by the predicate *abstr0:*

```
abstr0(V,X—>Y,cS—>AX—>AY) :- ! ,
      abstr0(V,X,AX) ,
      abstr0(V,Y,AY).
abstr0(V,V,cI) :- !.
abstr0(V,X,cK—>X).
```

V is a variable, *X* is an expression, and the third argument of *abstr0* is the expression with variable removed. So in the following:

```
abstr0(id(x),plus—>num(1)—>id(x),X).
```

we have

```
X = cS—>(cS—>(cK—>plus)—>(cK—>num(1)))—>cI
```

with no variables, and only the constants *plus* and *num(1),* the combinators and —>.

When the resulting expression is applied to an actual argument, these combinators, speaking anthropomorphically, place the actual arguments in the right places so that the evaluated result is the same as would be given (by extensionality) by evaluating the original expression with variables and by making the appropriate substitutions of actual arguments for variables. The advantage of not using variables, of course, is that an environment is not necessary and that no substitution algorithm is necessary.

It is clear, however, that this abstraction specification albeit elegant, leads to expressions much longer than the original. It is possible, however, to control the size of the resulting expression by introducing combinators which are "optimizing" in the sense that if a variable which is being abstracted is not used in the original expression, then the resulting expression will not have any redundancies. Some of these optimizing combinators are described in [Burge,1975]; a more effective set was introduced by Turner who also extended the notion of abstraction of variables to a context in which there was a primitive operation of pair construction in addition to the primitive operation of function application.

The predicate for abstraction in the specification of HASL's semantics is based on Turner's technique: *abstract* specifies how a list of variables is to be removed; *abstr* specifies how a single variable is removed; and *combine* specifies the optimizations which control the size of the resulting expressions.The first argument to *abstract* is a list uncurrying combinator which splits a structure into its components, and is an aspect of HASL's (restricted) unification. If a formal argument on the left hand side of a clausal definition is being "opened up", the combinator (cU_s) is strict: if the actual argument does not have the appropriate list structure then the value *fail* must result; in other cases, the list uncurrying combinator (cU) need not be strict.

Since constants may be HASL arguments, the abstraction predicate must specify what the resulting expression ought to be: in a strict position, removing a constant from an expression *E* means that when the resulting expression is applied to an actual argument, that argument must match exactly the removed constant, and so the parse tree is modified from *E* to *cMATCH* ---> *X* ---> *E* where *X* is the constant being abstracted; otherwise the resulting tree is *cK*--->*E*.

We may now examine the *semantic* relation in detail. The *semantic* relation specifies a traversal of the parse tree which results in a new tree from which all identifiers except global identifiers have been removed. For a subtree of the form *X : Y* or *X* ---> *Y,* the resulting tree is specified by:

```
semantic(X:Y,Sx:Sy) :- semantic(X,Sx), semantic(Y,Sy).
semantic(X—>Y,Sx—>Sy) :- semantic(X,Sx), semantic(Y,Sy).
```

Related to a subtree of the form *where(Exp,Defs)* is a subtree *Combinators* specified by

```
semantic(where(Exp,Defs),Combinators) :- abstract_locals(where(Exp,Defs),Combinators).
```

The predcate *abstract_locals* reforms the *where* node into a subtree of the form *AbsE* ---> *(cY* ---> *AbsD):*

```
abstract_locals(where(Exp,Defs),AbsE—>(cY—>AbsD)) :-
comp_defs(Defs,Ids,Abs),
abstract(cU,Ids,Exp,AbsE),
abstract(cU,Ids,Abs,AbsD).
```

cY is HASL's fixed point combinator whose reduction is defined as

```
cY—>X =>> Res :- X —> (cY —> X) =>> Res.
```

This is read as: *cY--->X* reduces to *Res* if *X--->(cY--->X)* reduces to *Res*. In the *abstract_locals* predicate, *Defs* are compiled by *comp_defs* to a list of identifiers defined *(Ids)* and a list of defined expressions from which all local variables have been removed *(Abs)*. The list of variables is abstracted from *Exp* and from *Abs*, specifying the subtrees *AbsE* and *AbsD*, respectively. The abstraction of *Ids* from *Abs* is the method of implementing mutually recursive definitions.

The predicate *comp_defs* builds the list of identifiers and abstractions by compiling each definition in *Def* using the predicate *comp_def*. A definition of arity 0 is left unchanged by the first clause of *comp_def*. As was mentioned in Section 2, the clauses defining a function are compiled as if one large unification expression had been specified. This compilation is specified by the predicate *comp_func*. The variables which are introduced by *comp_func* are of the form *id(1),id(2)*, etc., (these are not HASL variables) and must later be abstracted from *Code0* which is returned by *comp_func* to yield the *Code* tree for a definition:

```
comp_def(def(Name,0,Def),def(Name,0,Def)) :- !.
comp_def(def(Name,Arity,Funcs),def(Name,Arity,Code)) :-
Arity > 0,
comp_func(Funcs,Arity,Code0),
generate_seq(Arity,Ids),
abstract(cU_s,Ids,Code0,Code).
```

The predicate *generate_seq* specifies a relation between *Arity* and the list of introduced identifiers *Ids* which later gets removed!

A function is compiled clause by clause in reverse order. The last clause of any function is compiled by *comp_func* to

```
cCONDF —> Abs —> fail
```

where *Abs* is variable-free. *cCONDF* is a combinator defined as follows:

```
cCONDF —> X —> Y =>> Res :-
X =>> Rx, !,
cond_fail(Rx,Y,Res).
```

and is read: *cCONDF--->X--->Y* reduces to *Res* if *X* reduces to *Rx* and if *Rx* is not *fail* as determined by *cond_fail;* otherwise, *cond_fail* specifies that the value of *Res* is the value of the reduction of *Y*.

Remaining clauses defining a function are compiled by *comp1_func* to:

```
cCONDF —> Abs —> Sofar
```

where *Abs* is the compiled clause and *Sofar* is the code for the clauses already compiled.

A clause is compiled by the predicate *comp_clause:*

```
comp_clause(func(Fseq,Exp),Arity,Abs) :-
note_repeats(Fseq,MarkedFseq),
semantic(Exp,Sexp),
abstract(cU_s,MarkedFseq,Sexp,Aps),
generate_applies(Aps,Arity,Abs).
```

[1] The predicate *note_repeats* relates a list of formals, *Fseq*, to a marked list of formals *MarkedFseq*, where the second, third, etc., occurrences of a formal identifier *id(x)* have been replaced by *match(id(x))*. When *id(x)* is eventually abstracted from the right-hand side of a clause, this insures - by unification - that each occurrence of *id(x)* is matched to the same value. In the definition of *member* for example,

member a (a:x) = true

both occurrences of *a* must be bound to the same value. The *abstract* predicate treats repeated occurrences of an identifier in the way it treats constants.

[2] *Exp* is related by the *semantic* relation to *Sexp*.

[3] The marked formal sequence is abstracted from *Sexp* to yield *Aps*.

[4] The identifiers *id(1)*, *id(2)*, etc., are introduced.

The interested reader may follow on his own the specification of the *semantic* relation for subtrees labeled by the functor *unify* and for trees rooted at the functor *global*. It only needs to be said that a global definition such as

def suc x = 1 + x;

results in the following clause being added to HASL's database:

global(suc,cC1—>cCONDF—>(cADD—>num(1))—>fail).

Global names in any HASL expression are replaced at reduction time by their value as specified by the second component of *global*.

Some comments are due about the way we have compiled clauses into a function. In SASL, Turner allowed different clauses defining a function to have different arities. For example:

f 0 b = c
f 1 = d
f x y z = e

Thus, when an application of *f* is encountered in a SASL expression, it is impossible to know in advance, ie, at compile time, how much of the SASL expression to the right of *f* would actually be used by *f*. To cope with this, Turner introduced what he called a combinator "TRY, with rather peculiar reduction rules" [Turner,1981]. We had earlier implemented SASL in Prolog, and the specification of TRY in logic caused an enormous amount of trouble: it seemed to require at reduction time a stack to hold everything to the right of *f* in a SASL expression (ie, either to the end of the SASL expression, or to the first right parenthesis). The TRY combinator itself seemed to come in two arities: one of arity 3 for stacking everything to the right to be passed to each clause to be tried; and one of arity 2 to attempt clauses in order to find the applicable one. No other combinator seemed to require this explicit stack, but at reduction time the stack had to be passed as part of the state of the reduction to each combinator rule in case some clausally defined function were invoked. The presence of the stack in the logical specification seemed too operational and too distasteful, and there seemed no way to write the SASL reduction rules completely without it. This may have been simply a result of our confusion; or more profoundly a case where Wittgenstein's dictum held: *Was sich ueberhaupt sagen laesst, laesst sich klar sagen; und wovon man nicht reden kann, darueber muss man schweigen.* At any rate, HASL was born partly as a result of the hassle of trying to clean up the SASL reduction machine.

The cCONDF combinator was introduced to deal with a kind of conditional expression which arises often in dealing with unification based conditional binding expressions and in applying clausally defined functions: we could simply use the cCOND combinator, but the resulting code would be longer. Either way is simpler and clearer than using the TRY combinator! It should finally be noted that the restriction that all clauses defining a function have the same arity - which makes use of the cCONDF combinator feasible for compiling functions - imposes no loss of generality on what can be expressed in HASL: the sole interesting example in [Turner,1981]

which makes use of different arities can be expressed without utilizing clauses of different arities.

7. The Specification of HASL Reduction.

The specification of the HASL reduction relation consists mainly of a set of rules as to how the HASL combinators are reduced. The combinator cS, for example, introduced in the previous section, is reduced as follows:

```
cS —> Z —> Y —> X =>> Res :-
Z —> X —> (Y —> X) =>> Res.
```

Here, "=>>" is the infix reduction operator. The above specification is read: *cS ---> Z ---> Y ---> X* reduces to *Res* if *Z ---> X ---> (Y ---> X)* reduces to *Res*.

Associated with each combinator is an *arity*, for example:

```
arity(cS,3)
```

which indicates the number of arguments necessary for the reduction to take place. An expression such as

```
cS —> X —> Y
```

cannot be further reduced as it is already in head normal form. The reduction rules are listed in order of increasing arity; at the end of each group of rules for a given arity, there is a rule such as:

```
C —> X —> Y =>> C —> X —> Y :-
arity(C,D) , D >= 3 , !.
```

which would specify that *cS ---> X ---> Y* is already in head normal form.

The general reduction rule is to reduce the leftmost node of the combinator tree (the leftmost redex); if that node has not been reached, none of the combinator reduction rules apply. To handle the case of moving to the leftmost redex, the following (last but one) reduction rule applies:

```
X —> Y —> Z =>> Res :-
X —> Y =>> Rxy,
not same(X—>Y,Rxy),
Rxy —> Z =>> Res.
```

The reduction rules are recursively applied to try and reduce *X ---> Y* to head normal form; if *X ---> Y* is not in head normal form, then *Rxy* is head normal form for *X ---> Y* and *Rxy ---> Z* is reduced to *Res*.

The last reduction rule *X =>> X* specifies that *X* is already in head normal form.

Some combinators, such as the combinator *cCONDF*, defined in the last section, recursively call on the reduction machine. So does the combinator *cMATCH* which specifies unification:

```
cMATCH —> X —> Y —> Z =>> Rees :-
X =>> Redx, !,
Z =>> Redz, !,
eqnormal(Redx,Redz,Y,fail,Req),
Req =>> Res.
```

X and *Z* are reduced to Redx and Redz, respectively, and if they have the same normal form, *Y* is unified with *Req* and is reduced to *Res;* otherwise, *fail* is unified with *Req* and a trivial reduction reduces the entire match to *fail*. The binding of arguments to HASL formal variables - another part of HASL's restricted unification - is accomplished at the reduction stage by the combinators simply placing rhe actual arguments in their proper places for evaluation!

HASL numbers, truth values, and characters are tagged by the functors *num, logical* and *char*. (Lists are tagged by :.) Various parts of the reduction machine use these functors for type checking. For example, the addition component of the "arithmetic unit" specifies that addition is strict:

```
add(num(X),num(Y),num(Z)) :- Z is X + Y, !.
add(X,Y,fail).
```

HASL type checking functions such as *number* are defined globally and apply type checking combinators such as

```
cNUMBER --> X =>> Res :- type_check(num(X),Res).
```

The predicate *type_check* is specified by:

```
type_check(Form,logical(true)) :- Form, !.
type_check(Form,logical(false).
```

The reduction from head normal form to normal form is specified by the relation >>> whcih also has the side effect of printing the value of the original HASL expression in an appropriate format.

8. Applications, Conclusions, Further Work.

[1] One of our interests is in building a logical translator writing system based on Scott-Strachey denotational semantics. The general idea is to use DCGs for lexical and syntactic analysis and to produce an applicative expression which denotes the "value" of a program. The applicative expression must then be reduced to its value. It is our intent to construct the system so that HASL expressions are used as the applicative expressions which denote the values of programs.

Peter Mosses [Mosses,1979] Semantics Implementation System (SIS, implemented in BCPL) provides a "hard-wired" model for this project. It allows one to specify a grammar and the denotational semantics of a language, and produces for any program in that language an applicative expression in a language called DSL which is a slightly sugared version of a lambda calculus language LAMB. As a first step in our project we will probably compile DSL expressions to combinators and use the HASL reduction machine to reduce DSL expressions to the values which they denote.

[2] HASL may be thought of as a functional sublanguage of Prolog. More generally, we can think of a deduction machine (eg, Prolog) which has a reduction machine (eg, HASL) as a component. Another model here is LOGLISP [Robinson&Siebert,1980] in which LOGIC is the deduction machine and LISP is the reduction machine. In the case of LOGLISP, however, it took quite a lot of work to define a suitable reduction mechanism for LISP: the notion of reduction of LISP expressions is fairly complex and is not identical to evaluation of LISP expressions. We suggest that since HASL is defined in terms of a notion of reduction *ab initio,* it is simpler and perhaps cleaner mathematically to consider a deduction-reduction machine with HASL as the reduction component. LOGLISP, however, treats the LOGIC machine and the LISP machine as equal components able to call on each other for computations; it remains to be investigated how HASL might call on the deduction machine.

[3] The HASL reduction machine has some notion of partial evaluation. If one defines

```
def f cond a b = cond -> a ; b:
```

then *f true* is the function which when applied to two arguments selects the first one. In terms of combinators, the reduction of *f true* is:

```
cC—>(cB1—>(cB1—>cC1)—>cCONDF—>
        cCOND—>logical(true))—>fail
```

Another observation is that the abstraction of variables from an expression is a relation between a variable, an expression, and another expression without that variable. The abstraction may be run "backwards" and a variable may be put into a variable-free combinator expression to get something more readable. For example, in:

```
abstr0(id(x),E,cS—>(cS—>(cK—>plus)—>(cK—>num(1)))—>cI).
```

we have

```
E = plus—>num(1)—>id(x).
```

HASL abstraction is more complicated than this, but in principle one may think of decompiling variable-free expressions.

One might think of combining these two observations to get a notion for a debugging method for applicative languages: a partially evaluated expression may have some variables put back into it, and then one might try using the lexical and syntactic DCGs as generators rather than as recognizers to produce a readable HASL expression.

It may not be entirely frivolous to think in fact of generating programs which compute a given value. The reduction relation may be run backward to derive combinator strings which could be translated into HASL expressions. Of course there are infinitely many such expressions and most of them are trivial and/or uninteresting. Could one place constraints on the searching of the space of HASL expressions which compute a given value to produce interesting expressions?

[4] Pragmatically, Prolog is ideal for designing and testing experimental languages. One tends not to carry out language experiments other than on paper - or in one's head - if implementation requires extensive coding in a low level language. But - the HASL interpreter described here, implemented in CProlog to run on a VAX 780 under Berkeley UNIX, is *slow*. A Prolog compiler which optimizes tail recursion and runs under UNIX is an absolute necessity.

9. Acknowledgements.

This work was supported by the National Science and Engineering Research Councilof Canada. I must also thank the UBC Laboratory for Computational Vision for time on its VAX running Berkeley Unix: modern and adequate computing facilities are not currently made available to the computer science department by UBC.

10. References.

[Abramson,1982a]

Abramson, H., *Unification-based Conditional Binding Constructs,* Proceedings First International Logic Programming Conference, Marseille, 1982.

[Abramson,1982b]

Abramson, H., *A Prolog Implementation of SASL,* Logic Programming Newsletter 4, Winter 1982/1983.

[Burge,1975]

Burge, W.H., *Recursive Programming Techniques,* Addison-Wesley, 1975.

[Colmerauer,1978]

Colmerauer, A., *Metamorphosis Grammars,* in Natural Language Communication with Computers, Lecture Notes in Computer Science 63, Springer, 1978.

[Henderson&Morris,1976]

Henderson, P. & Morris, J.H., *A lazy evaluator,* Conference Record of the 3rd ACM Symposium on Principles of Programming Languages, pp. 95-103, 1976.

[Moss,1979]

Moss, C.D.S., *A Formal Description of ASPLE Using Predicate Logic,* DOC 80/18, Imperial College, London.

[Moss,1981]

Moss, C.D.S., *The Formal Description of Programming Languages using Predicate Logic,* Ph.D. Thesis, Imperial College, 1981.

[Moss,1982]

Moss, C.D.S., *How to Define a Language Using Prolog,* Conference Record of the 1982 ACM Symposium on Lisp and Functional Programming, Tittsburgh, Pennsylvania, pp. 67-73, 1982.

[Mosses,1979]

Mosses, Peter, *SIS - Semantics Implementation System: Reference Manual and User Guide,* DAIMI MD-30, Computer Science Department, Aarhus University, Denmark, 1979.
[Pereira&Warren,1980]

Pereira, F.C.N. & Warren, D.H.D, *Definite Clause Grammars for Language Analysis,* Artificial Intelligence, vol. 13, pp. 231-278, 1980.

[Robinson&Siebert,1980a]

Robinson, J.A. & Siebert, E.E., *LOGLISP - an alternative to Prolog,* School of Computer and Information Science, Syracuse University, 1980.

[Robinson&Siebert,1980b]

Robinson, J.A. & Siebert, E.E., *Logic Programming in LISP,* School of Computer and Information Science, Syracuse University, 1980.

[Turner,1976]

Turner, D.A., *SASL Language Manual,* Department of Computational Science, University of St. Andrews, 1976, revised 1979.

[Turner,1979]

Turner, D.A., *A new implementation technique for applicative languages,* Software - Practice and Experience, vol. 9, pp. 31-49.

[Turner,1981]

Turner, D.A., *Aspects of the Implementation of Programming Languages: The Compilation of an Applicative Language to Combinatory Logic,* Ph.D. Thesis, Oxford, 1981.

[Warren,1977]

Warren, David H.D., *Logic programming and compiler writing,* DAI Research Report 44, University of Edinburgh, 1977.

21

```
/* lexical rules */

reserved(true,constant(logical(true))).
reserved('TRUE',constant(logical(true))).
reserved(false,constant(logical(false))).
reserved('FALSE',constant(logical(false))).
reserved(fail,constant(fail)).
reserved('FAIL',constant(fail)).
reserved(def,def).
reserved('DEF',def).
reserved(where,where).
reserved('WHERE',where).

lexemes(X)    -> space , lexemes(X).
lexemes([X|Y]) -> lexeme(X) , lexemes(Y).
lexemes([])   -> [].


lexeme(Token) ->
      word(W) , ! , { name(X,W) , ( reserved(X,Token) ; id(X) = Token ) }.
lexeme(constant(Con)) -> constant(Con) , !.
lexeme(Punct)         -> punctuation(Punct) , !.
lexeme(op(Pr,Comb))   -> op(Pr,Comb) , !.

space -> " " , !.
space -> [10], !. /* carriage return */

num(num(N)) -> number(Number) , ! , { name(N,Number) }.

number([D|Ds]) -> digit(D) , digits(Ds).

digit(D) -> [D] , { D>47, D<58 }. /* 0...9 */

digits([D|Ds]) -> digit(D) , digits(Ds).
digits([])    --> [].

word([L|Ls]) -> letter(L) , lords(Ls).

letter(L) -> [L] , { (L>96,L<123 ; L>64,L<90) }. /* a-z, A-Z */

lords([L|Ls]) -> ( letter(L) ; digit(L) ) , lords(Ls).
lords([])    -> [].

/* in op(N,O) N designates the binding power of the operator O. */
op(0,cAPPEND) -> "++" , !.
op(0,cCONS)   -> ":" , !.
op(1,cOR)     -> "|" , !.
op(2,cAND)    -> "&" , !.
op(3,cLSE)    -> "<=" , !.
op(3,cGRE)    -> ">=" , !.
op(3,cNEQ)    -> "=" , !.
op(3,cEQ)     -> "=" , !.
op(3,cGR)     --> ">" , !.
```

22

```
op(3,cLS)    -> "<" , !.
op(4,cADD)   -> "+" , !.
op(4,cSUB)   -> "-" , !.
op(5,cMULT)  -> "*" , !.
op(5,cDIV)   -> "/" , !.
op(6,cB)     -> "." , !.

hasl_string(C:Cs) -> stringchar(C) , hasl_string(Cs).
hasl_string(nil)  -> [].

hasl_char(C) -> "%" , stringchar(C) , !.


stringchar(char(A))   -> [C] , { C =\= 39 , name(A,[C]) } , !.
stringchar(char(''')) -> "'" , !.

string(S) -> "'" , hasl_string(S) , "'" ,!.

constant(N)  -> num(N) , !.
constant(C)  -> hasl_char(C) , !.
constant(S)  -> string(S) , !.
constant(nil) -> "[]" , !.

punctuation(tilde)        -> "~" , !.
punctuation(comma)        -> "," , !.
punctuation(lparen)       -> "(" , !.
punctuation(rparen)       -> ")" , !.
punctuation(condarrow)    -> "->" , !.
punctuation(rightcrossbow) -> "-}", !.
punctuation(leftcrossbow) -> "{-", !.
punctuation(lbrack)       -> "[", !.
punctuation(rbrack)       -> "]", !.
punctuation(unifyarrow)   -> "=>", !.
punctuation(semicolon)    -> ";" , !.

/* The following predicates constitute the interface
  between lexical and syntactic analysis. Predicates
  with names starting with 't', eg, tCOLON, are the
  terminals in syntactic analysis.
*/

tCOLON        -> [op(0,cCONS)].
tPLUSPLUS     -> [op(0,cAPPEND)].
tCOMMA        -> [comma].
tLBRACK       -> [lbrack].
tRBRACK       -> [rbrack].
tLPAREN       -> [lparen].
tRPAREN       -> [rparen].
tUNIFYARROW   -> [unifyarrow].
tLEFTCROSSBOW -> [leftcrossbow].
tRIGHTCROSSBOW -> [rightcrossbow].
tCONDARROW    -> [condarrow].
tEQUAL        -> [op(3,cEQ)].
tSEMICOLON    -> [semicolon].
```

```
tWHERE          -> [where].
tDEF          -> [def].
tNOT          -> [tilde].
tNEGATE         -> [op(4,cSUB)].
tPLUS         -> [op(4,cADD)].
tIDENT(id(Id)) -> [id(Id)].
tCONSTANT(C)  -> [constant(C)].
tOP(Pr,Comb)  -> [op(Pr,Comb)].
```

```
/* syntactic rules */

def(global(Ds)) -->
      tDEF , defs(Ds) , tSEMICOLON.


definition(def(Id,Arity,func(Fseq,Exp))) -->
      tIDENT(Id) , fseq(Fseq) , ! , tEQUAL , expression(Exp) ,
      { seq_length(Fseq,Arity) }.

definition(Def) -->
      formal(Formal) , ! , tEQUAL , expression(Exp) ,
      { expandef(def(Formal,0,Exp),Def) }.

defs(Ds) --> definition(D) , ! , { append_def(D,[],Deflist) } ,
          defs1(Deflist,Ds).

defs1(D,Ds) --> tCOMMA , definition(Def) , ! ,
            { mergedef(D,Def,Dm) } , defs1(Dm,Ds).
defs1(D,D)  --> [].

fseq(Fseq) --> formal(Formal) , ! , fseq1(Formal,Fseq).

fseq1(F1,[F1|F]) --> formal(F2) , ! , fseq1(F2,F).
fseq1(F,F) --> [].

formal(Id) --> tIDENT(Id) , !.
formal(const(C)) --> tCONSTANT(C) , !.
formal(flist(Flist)) --> tLBRACK , flist(Flist) , ! , tRBRACK.
formal(flist(Flist)) --> tLPAREN , fprimary(Flist) , ! , tRPAREN.

flist(F1:F2) --> fprimary(F1) , ! , flist1(F2).
flist(const(nil)) --> [].

flist1(F) --> tCOMMA , flist(F).
flist1(const(nil)) --> [].

fprimary(F) --> formal(F1) , ! , fprimary1(F1,F).

fprimary1(F1,F1:F) --> tCOLON , formal(F2) , ! , fprimary1(F2,F).
fprimary1(F,F) --> [].

expression(E) --> def(E).
expression(E) --> unification(E1) , ! , expression(E1,E).

expression(E1,where(E1,Ds)) --> tWHERE , defs(Ds).
expression(E,E) --> [].

unification(unify(Fseq,E1,E2,E3)) -->
      fseq(Fseq) , tLEFTCROSSBOW , expression(E1) , tUNIFYARROW ,
        expression(E2) , tSEMICOLON , expression(E3).

unification(U) --> condexp(U).
```

```
condexp(E) -> exp1(E1,0) , ! , condexp1(E1,E).

condexp1(E1,cCOND --> E1 --> E2 --> E3) ->
      tCONDARROW , expression(E2) , ! , tSEMICOLON , condexp(E3).
condexp1(E1,unify(Fseq,E1,E2,E3)) ->
      tRIGHTCROSSBOW , fseq(Fseq) , tUNIFYARROW ,
        expression(E2) , tSEMICOLON , expression(E3).
condexp1(E,E) -> [].

exp1(E,P) -> tPLUS , exp1(E1,6) , ! , exp2(E1,E,P).
exp1(E,P) -> tNEGATE , exp1(E1,6) , ! , exp2(cNEGATE --> E1,E,P).
exp1(E,P) -> tNOT , exp1(E1,3) , ! , exp2(cNOT --> E1,E,P).
exp1(E,P) -> comb(E1) , ! , exp2(E1,E,P).

/* since : or cons is a primitive in HASL: */
exp2(E1,E,0) -> tCOLON , exp1(E2,0) , ! , exp2(E1 : E2,E,1).

/* since + + or append is the only other zero level operator: */
exp2(E1,E,0) -> tPLUSPLUS , exp1(E2,0) , ! , exp2(cAPPEND --> E1 --> E2,E,1).

/* : and + + are right associative; all others are left associative: */
exp2(E1,E,P) -> tOP(Q,Op) , { P < Q } , ! , exp1(E2,Q) ,
            exp2(Op --> E1 --> E2,E,P).

exp2(E,E,P) -> [].

comb(C) -> primary(P) , ! , comb1(P,C).

comb1(P1,C) -> primary(P) , ! , comb1(P1 --> P,C).
comb1(C,C) -> [].

primary(L) --> tLBRACK , explist(L) , ! , tRBRACK.
primary(I) -> tIDENT(I) , !.
primary(C) -> tCONSTANT(C) , !.
primary(E) --> tLPAREN , expression(E) , ! , tRPAREN.


explist(E1 : E2) -> exp1(E1,0) , ! , explist1(E2).
explist(nil) -> [].
explist1(E) -> tCOMMA , explist(E).
explist1(nil) -> [].

/* The following predicates are used to check that each clause
   defining a function has the same arity, and to merge all
   definitions made at the same time into a single list of
   definitions.
*/

mergedef(Deflist,Def,Defmerge) :-
      flat(Def,FlatDef) ,
      merge(Deflist,FlatDef,Defmerge).

merge(Deflist,[Def|Defs],Defmerge) :-
    merge(Deflist,Def,Deflist1) ,
```

```
    merge(Deflist1,Defs,Defmerge).

merge([def(id(X),0,D)|Deflist],
      def(id(X),0,D1),
      [def(id(X),0,D)|Deflist]) :-
      write(X) ,
      write(' is a constant already defined: ') ,
      write(D) , nl ,
      write('definition ignored: ') ,
      write(D1) , nl.
merge([def(id(X),N,D)|Deflist],
      def(id(X),N,D1),
      [def(id(X),N,[D1|D])|Deflist]) :- !.
merge([def(id(X),N,D)|Deflist],
      def(id(X),M,D1),
      [def(id(X),N,D)|Deflist]) :-
      write('wrong number of arguments in definition of:') ,
      write(def(id(X),M,D1)) ,
      write('should be ') , write(N) , nl.
merge([def(id(Y),M,Dy)|Deflist],
      def(id(X),N,D),
      [def(id(Y),M,Dy)|Deflist]) :-
      defined(X,Deflist,Dx) , ! ,
      write(X) ,
      write(' already defined: ') ,
      write(Dx) , nl ,
      write(def(id(X),N,D)) ,
      write(' ignored.') , nl.
merge(Deflist,
      Def,
      [Def|Deflist]).

defined(Y,[def(id(Y),_,Dy)|_],Dy).
defined(Y,[def(id(X),_,_)|Deflist],Dy) :-
    defined(Y,Deflist,Dy).

seq_length([F|G],N) :- ! ,
        seq_length(G,M) ,
        N is 1 + M.
seq_length(F,1).


append_def(def(A,B,C),Z,[def(A,B,C) |Z]) :- !.
append_def([X|Y],Z,[X|W]) :-
      append_def(Y,Z,W).

flat(def(X,Y,Z),def(X,Y,Z)).
flat([def(X,Y,Z)|Defs],[def(X,Y,Z)|FDefs]) :-
           flat(Defs,FDefs) , !.
flat([DefHd|DefTl],Flat) :-
      flat(DefHd,FlatHd) ,
      flat(DefTl,FlatTl) ,
      append_def(FlatHd,FlatTl,Flat).
```

27

```
expandef(Defs,Def) :-
      expand(Defs,Def1) ,
      flat(Def1,Def).

expand(def(flist(X:const(nil)),0,Exp),Defx) :-
      expand(def(X,0,Exp),Defx).
expand(def(flist(X:Y),0,Exp),[Defx|Defy]) :-
      expand(def(X,0,cHD --> Exp),Defx) ,
      expand(def(flist(Y),0,cTL --> Exp),Defy).
expand(def(flist(X),0,Exp),def(X,0,Exp)).
expand(def(F,0,Exp),def(F,0,Exp)).
```

```
/* semantic rules */

semantic(X:Y,Sx:Sy) :-
      semantic(X,Sx) ,
      semantic(Y,Sy).
semantic(X—>Y,Sx—>Sy) :-
      semantic(X,Sx) ,
      semantic(Y,Sy).
semantic(where(Exp,Defs),Combinators) :-
      abstract_locals(where(Exp,Defs),Combinators).
semantic(unify(Fseq,E1,E2,E3),cCONDF—>Exp—>Se3) :-
      semantic(E1,Se1) ,
      semantic(E2,Se2) ,
      semantic(E3,Se3) ,
      translate_unification(Fseq,Se1,Se2,Exp).
semantic(global(Defs),global) :-
      installdefs(Defs).
semantic(X,X).

abstract(U,nil,Abs,Abs).
abstract(U,flist(Flist),Exp,Abs) :-
      abstract(U,Flist,Exp,Abs).
abstract(U,[X|Y],E,Abs) :-
      abstract(U,Y,E,Abs1) ,
      abstract(U,X,Abs1,Abs).
abstract(U,id(X),E,Abs) :-
      abstr(id(X),E,Abs).
abstract(cU,const(X),E,cK —> E).
abstract(cU_s,const(X),E,cMATCH —> X —> E).
abstract(cU_s,match(X),E,cMATCH —> X —> E).
abstract(U,(X : Y),E,U —> Abs) :-
      abstract(U,Y,E,Abs1) ,
      abstract(U,X,Abs1,Abs).


abstr(V,X —> Y,Abs) :-
      abstr(V,X,AX) ,
      abstr(V,Y,AY) ,
      combine(—>,AX,AY,Abs) , !.
abstr(V,(X : Y),Abs) :-
      abstr(V,X,AX) ,
      abstr(V,Y,AY) ,
      combine(:,AX,AY,Abs) , !.
abstr(id(X),id(X),cI) :- !.
abstr(V,X,cK —> X).

combine(—>,cK —> X,cK —> Y,cK —> (X —> Y)).
combine(—>,cK —> X,cI,X).
combine(—>,cK —> (X1 —> X2),Y,cB1 —> X1 —> X2 —> Y).
combine(—>,cK —> X,Y,cB —> X —> Y).
combine(—>,cB —> X1 —> X2,cK —> Y,cC1 —> X1 —> X2 —> Y).
combine(—>,X,cK —> Y,cC —> X —> Y).
combine(—>,cB —> X1 —> X2,Y,cS1 —> X1 —> X2 —> Y).
combine(—>.X,Y,cS —> X —> Y).
```

```
combine(:,cK —> X,cK —> Y,cK —> (X : Y)).
combine(:,cK —> X,Y,cB_p —> X —> Y).
combine(:,X,cK —> Y,cC_p —> X —> Y).
combine(:,X,Y,cS_p —> X —> Y).

generate_seq(1,id(1)) :- !.
generate_seq(N,Y) :-
     N1 is N - 1 ,
     gen_seq(N1,id(N),Y).

gen_seq(1,X,[id(1)|X]) :- !.
gen_seq(N,X,Y) :-
      N1 is N - 1 ,
      gen_seq(N1,[id(N)|X],Y).

generate_applies(X,N,Y) :-
      generate_seq(N,Seq) ,
      gen_applies(X,Seq,Y).

gen_applies(X,[Hd|Tl],Y) :- ! ,
      gen_applies(X —> Hd,Tl,Y).
gen_applies(X,S,X —> S).

restructure(X —> (Y —> Z),W) :- restruct(X,Y —> Z,W).
restructure(X —> Y,X —> Y).

restruct(X,Y —> Z,A —> Z  ) :- restruct(X,Y,A).
restruct(X,Y,X —> Y).

comp_clause(func(Fseq,Exp),Arity,Abs) :-
      note_repeats(Fseq,MarkedFseq) ,
      semantic(Exp,Sexp) ,
      abstract(cU_s,MarkedFseq,Sexp,Aps) ,
      generate_applies(Aps,Arity,Abs).

comp_func([func(Fseq,Exp)|Funcs],Arity,Code) :-
      comp_clause(func(Fseq,Exp),Arity,Abs) ,
      comp1_func(Funcs,Arity,cCONDF —> Abs —> fail,Code) .
comp_func(func(Fseq,Exp),Arity,cCONDF —> Abs —> fail) :-
      comp_clause(func(Fseq,Exp),Arity,Abs).

comp1_func([func(Fseq,Exp)|Funcs],Arity,Sofar,Code) :-
      comp_clause(func(Fseq,Exp),Arity,Abs) ,
      comp1_func(Funcs,Arity,cCONDF —> Abs —> Sofar,Code).
comp1_func(func(Fseq,Exp),Arity,Sofar,cCONDF —> Abs —> Sofar) :-
      comp_clause(func(Fseq,Exp),Arity,Abs).

comp_def(def(Name,0,Def),def(Name,0,Def)) :- !.
comp_def(def(Name,Arity,Funcs),def(Name,Arity,Code)) :-
      Arity > 0 , ! ,
      comp_func(Funcs,Arity,Code0) ,
      generate_seq(Arity,Ids) ,
      abstract(cU_s,Ids,Code0,Code).
```

```
comp_defs([Def|Defs],Ids,Abs) :-
      comp_def(Def,def(id(Id),Arity,Abs1)) ,
      comp_defs1(Defs,id(Id),Abs1,Ids,Abs).

comp_defs1([],Ids,Abs,Ids,Abs).
comp_defs1([Def|Defs],IdsIn,AbsIn,Ids,Abs) :-
      comp_def(Def,def(id(Id),Arity,Abs1)) ,
      comp_defs1(Defs,id(Id):IdsIn,Abs1:AbsIn,Ids,Abs).

abstract_locals(where(Exp,Defs),AbsE —> (cY —> AbsD)) :-
      comp_defs(Defs,Ids,Abs) ,
      abstract(cU,Ids,Exp,AbsE) ,
      abstract(cU,Ids,Abs,AbsD).

translate_unification(Fseq,E1,E2,Exp) :-
      note_repeats(Fseq,MarkedFseq) ,
      abstract(cU_s,MarkedFseq,E2,Abs) ,
      restructure(Abs —> E1,Exp).

installdefs(Defs) :-
       comp_defs(Defs,Ids,Abs) ,
       install(Ids,Abs).

install(id(Id):Ids,Def:Defs) :-
      global(Id,DefId) , ! ,
      write(Id) , write(' already globally defined.') ,
      write(' New definition ignored.') , nl ,
      install(Ids,Defs).
install(id(Id):Ids,Def:Defs) :-
      assertz(global(Id,Def)) ,
      install(Ids,Defs).
install(id(Id),Abs) :-
      global(Id,DefId) , ! ,
      write(Id) , write(' already globally defined.') ,
      write(' New definition ignored.') , nl.
install(id(Id),Def) :-
      assertz(global(Id,Def)).

member(Id,[Id|_]).
member(Id,[_|Ids]) :- member(Id,Ids).

note_repeats(Fseq,Marked) :-
          mark_repeats(Fseq,[],_,Marked).

mark_repeats(id(Id),In,In,match(id(Id))) :-
          member(Id,In) , !.
mark_repeats(id(Id),In,[Id|In],id(Id)).
mark_repeats(flist(Flist),In,Out,flist(Marked)) :-
          mark_repeats(Flist,In,Out,Marked).
mark_repeats(Hd:Tl,In,Out,MarkedHd:MarkedTl) :-
          mark_repeats(Hd,In,In1,MarkedHd) ,
          mark_repeats(Tl,In1,Out,MarkedTl).
mark_repeats([Hd|Tl],In,Out,[MarkedHd|MarkedTl]) :-
          mark_repeats(Hd,In,In1,MarkedHd) ,
```

```
        mark_repeats(Tl,In1,Out,MarkedTl).
mark_repeats([],In,In,[]).
mark_repeats(const(C),In,In,const(C)).
```

```
/* reduction rules */

id(X) --> id(Y) =>> Res :-
        global(X,DefX) ,
        global(Y,DefY) ,
        DefX --> DefY =>> Res.

id(X) =>> Def :-
        global(X,Def) , !.

id(X) =>> _ :-
        nl ,
        write('not defined: ') ,
        write(X) , nl , abort.

cI --> X =>> Res :-
        X =>> Res.

cY --> X =>> Res :-
        X --> (cY --> X) =>> Res.

cHD --> (X : Y) =>> Res :- ! ,
        X =>> Res.

cHD --> X =>> Res :-
        X =>> (Hd : Tl) , ! ,
        Hd =>> Res.

cTL --> (X : Y) =>> Res :- ! ,
        Y =>> Res.

cTL --> X =>> Res :-
        X =>> (Hd : Tl) , ! ,
        Tl =>> Res.

cCHAR --> X =>> Res :-
        type_check(char(X),Res).

cFAILURE --> X =>> Res :-
        type_check(failure(X),Res).

cLOGICAL --> X =>> Res :-
        type_check(logical(X),Res).

cFUNCTION --> X =>> Res :-
        type_check(function(X),Res).

cNUMBER --> X =>> Res :-
        type_check(num(X),Res).

cNOT --> X =>> Res :-
        X =>> Rx , ! ,
        choose(Rx,logical(false),logical(true),Res).
```

```
cNEGATE --> X =>> Res :-
        arith(sub,num(0),X,Res).

num(X) --> Y =>> num(X).

logical(X) --> Y =>> logical(X).

char(X) --> Y =>> char(X).

nil --> X =>> nil.

C --> X =>> C --> X :-
        arity(C,D) ,
        D >= 2 , !.

id(X) --> Y =>> Res :-
        global(X,Def) , ! ,
        Def --> Y =>> Res.

id(X) --> Y =>> Res :-
        nl ,
        write('not defined: ') ,
        write(X) , nl , abort.


Y --> id(X) =>> Res :-
        global(X,Def) , ! ,
        Y --> Def =>> Res.

Y --> id(X) =>> Res :-
        nl ,
        write('not defined: ') ,
        write(X) , nl , Y --> fail =>> Res.

(X : Y) --> num(1) =>> Res :- ! ,
        X =>> Res.

(X : Y) --> num(Z) =>> Res :- ! ,
        Z > 1 ,
        Z1 is Z - 1 ,
        Y --> num(Z1) =>> Res.

(X : Y) --> Z =>> Res :-
        Z =>> num(Num) , ! ,
        X : Y --> num(Num) =>> Res.

cK --> X --> Y =>> Res :-
        X =>> Res.

cU --> X --> Y =>> Res :-
        X --> (cHD --> Y) --> (cTL --> Y) =>> Res.

cU_s --> X --> (Y : Z) =>> Res :- ! ,
        X --> Y --> Z =>> Res.
```

```
cU_s —> X —> Y =>> Res :-
        Y =>> (Hd : Tl) , ! ,
        X —> Hd —> Tl =>> Res.
cU_s —> X —> Y =>> fail.

cAND —> X —> Y =>> Res :-
        X =>> Rx , ! ,
        choose(Rx,Y,logical(false),Res).

cOR —> X —> Y =>> Res :-
        X =>> Rx , ! ,
        choose(Rx,logical(true),Y,Res).

cEQ —> X —> Y =>> logical(Res) :-
        X =>> Rx , ! ,
        Y =>> Ry , ! ,
        eqnormal(Rx,Ry,true,false,Res).

cNEQ —> X —> Y =>> logical(Res) :-
        X =>> Rx , ! ,
        Y =>> Ry , ! ,
        eqnormal(Rx,Ry,false,true,Res).

cAPPEND —> nil —> Z =>> Z :- !.
cAPPEND —> (X : Y) —> Z =>> (X : Res) :- ! ,
        cAPPEND —> Y —> Z =>> Res.
cAPPEND —> X —> Y =>> Res :-
        X =>> Resx , ! ,
        not same(X,Resx) ,
        cAPPEND —> Resx —> Y =>> Res.

cLSE —> X —> Y =>> Res :-
        cNOT —> (cGR —> X —> Y) =>> Res , !.

cGRE —> X —> Y =>> Res :-
        cNOT —> (cLS —> X —> Y) =>> Res , !.

cLS —> X —> Y =>> Res :-
        arith(ls,X,Y,Res) , !.

cGR —> X —> Y =>> Res :-
        arith(gr,X,Y,Res) , !.

cADD —> X —> Y =>> Res :-
        arith(add,X,Y,Res) , !.

cSUB —> X —> Y =>> Res :-
        arith(sub,X,Y,Res) , !.

cMULT —> X —> Y =>> Res :-
        arith(mult,X,Y,Res) , !.

cDIV —> X —> Y =>> Res :-
        arith(div,X,Y,Res) , !.
```

```
cCONDF —> X —> Y =>> Res :-
        X =>> Rx , ! ,
        cond_fail(Rx,Y,Res).

C —> X —> Y =>> C —> X —> Y :-
        arity(C,D) ,
        D >= 3 , !.

cCOND —> X —> Y —> Z =>> Res :-
        X =>> Resx , ! ,
        choose(Resx,Y,Z,Res).

cMATCH —> nil —> Y —> Z =>> Res :-
        match(nil,Z,Y,Res).
cMATCH —> num(X) —> Y —> Z =>> Res :-
        match(num(X),Z,Y,Res).
cMATCH —> char(X) —> Y —> Z =>> Res :-
        match(char(X),Z,Y,Res).
cMATCH —> logical(X) —> Y —> Z =>> Res :-
        match(logical(X),Z,Y,Res).
cMATCH —> X —> Y —> Z =>> Res :-
        X =>> Redx , ! ,
        Z =>> Redz , ! ,
        eqnormal(Redx,Redz,Y,fail,Req) , ! ,
        Req =>> Res.

cS_p —> X —> Y —> Z =>> Res :-
        (X —> Z) : (Y —> Z) =>> Res.

cB_p —> X —> Y —> Z =>> Res :-
        X : (Y —> Z) =>> Res.

cC_p —> X —> Y —> Z =>> Res :-
        X —> Z : Y =>> Res.

cS —> Z —> Y —> X =>> Res :-
        Z —> X —> (Y —> X) =>> Res.

cB —> X —> Y —> Z =>> Res :-
        X —> (Y —> Z) =>> Res.

cC —> X —> Y —> Z =>> Res :-
        X —> Z —> Y =>> Res.

C —> X —> Y —> Z =>> C —> X —> Y —> Z :-
        arity(C,D) ,
        D >= 4 , !.

cS1 —> W —> X —> Y —> Z =>> Res :-
        W —> (X —> Z) —> (Y —> Z) =>> Res.

cB1 —> W —> X —> Y —> Z =>> Res :-
        W —> X —> (Y —> Z) =>> Res.
```

```
cC1 —> W —> X —> Y —> Z =>> Res :-
          W —> (X —> Z) —> Y =>> Res.

X —> Y —> Z =>> Res :-
          X —> Y =>> Rxy ,
          not same(X —> Y,Rxy) , ! ,
          Rxy —> Z =>> Res.

X =>> X.

same(X,X).

choose(logical(true),Y,Z,Res) :-
          Y =>> Res , !.
choose(logical(false),Y,Z,Res) :-
          Z =>> Res , !.
choose(X,Y,Z,fail).

match(X,Y,Z,Res) :-
          Y =>> Ry , ! ,
          eqnormal(X,Ry,Z,fail,R) , ! ,
          R =>> Res , !.

eqnormal(X,Y,T,F,T) :-
          equals(X,Y) , !.
eqnormal(X,Y,T,F,F).

equals(num(X),num(X)).
equals(char(X),char(X)).
equals(logical(X),logical(X)).
equals(nil,nil).
equals((A : B),(X : Y)) :-
          A =>> Reda ,
          X =>> Redx ,
          equals(Reda,Redx) , ! ,
          equals(B,Y).

is_failure((X —> Y)) :-
          is_failure(X).
is_failure(fail).

is_function((X —> Y)) :- is_function(X).
is_function(X) :- arity(X,_).

arity(cI,1).
arity(cY,1).
arity(cV,1).
arity(cHD,1).
arity(cTL,1).
arity(cNOT,1).
arity(cFUNCTION,1).
arity(cCHAR,1).
arity(cLOGICAL,1).
arity(cNUMBER,1).
```

```
arity(cFAILURE,1).
arity(cK,2).
arity(cU,2).
arity(cU_s,2).
arity(cEQ,2).
arity(cNEQ,2).
arity(cAND,2).
arity(cOR,2).
arity(cAPPEND,2).
arity(cCONDF,2).
arity(cSUB,2).
arity(cADD,2).
arity(cMULT,2).
arity(cDIV,2).
arity(cGRE,2).
arity(cLSE,2).
arity(cLS,2).
arity(cGR,2).
arity(cS,3).
arity(cC,3).
arity(cB,3).
arity(cS_p,3).
arity(cCOND,3).
arity(cMATCH,3).
arity(cB_p,3).
arity(cC_p,3).
arity(cS,3).
arity(cS1,4).
arity(cB1,4).
arity(cC1,4).

type_check(Form,logical(true)) :- Form , !.
type_check(Form,logical(false)).

char(X) :- X =>> char(_).

logical(X) :- X =>> logical(_).

num(X) :- X =>> num(_).

failure(X) :- X =>> Rx , ! , is_failure(Rx).

list(X) :- id(list) —> X =>> logical(true).

function(X) :- X =>> Rx , ! , is_function(Rx).

add(num(X),num(Y),num(Z)) :- Z is X + Y , !.
add(X,Y,fail).

sub(num(X),num(Y),num(Z)) :- Z is X - Y , !.
sub(X,Y,fail).

mult(num(X),num(Y),num(Z)) :- Z is X * Y , !.
mult(X,Y,fail).
```

```
div(num(X),num(Y),num(Z)) :- Z is X / Y , !.
div(X,Y,fail).

eq(X,Y) :- X =:= Y.

gr(num(X),num(Y),logical(true)) :- X > Y , !.
gr(num(X),num(Y),logical(false)) :- !.
gr(X,Y,fail).

ls(num(X),num(Y),logical(true)) :- X < Y , !.
ls(num(X),num(Y),logical(false)) :- !.
ls(X,Y,fail).

cond_fail(X,Y,X) :- not is_failure(X) , !.
cond_fail(_,Y,Ry) :- Y =>> Ry.

arith(Operation,X,Y,Res) :-
        X =>> Rx ,
        Y =>> Ry ,
        arithop(Operation,Rx,Ry,Res).

arithop(add,X,Y,Z) :-
        add(X,Y,Z).
arithop(sub,X,Y,Z) :-
        sub(X,Y,Z).
arithop(mult,X,Y,Z) :-
        mult(X,Y,Z).
arithop(div,X,Y,Z) :-
        div(X,Y,Z).
arithop(ls,X,Y,Z) :-
        ls(X,Y,Z).
arithop(gr,X,Y,Z) :-
        gr(X,Y,Z).

/* reduce to normal form */

reducelist(nil,nil).
reducelist(Hd : Tl,Nhd : Ntl) :- write(',') ,
                    Hd =>> Redhd , ! ,
                    Redhd >>> Nhd ,
                    Tl =>> Redtl , ! ,
                    reducelist(Redtl,Ntl).

reducestring(nil,nil).
reducestring(char(C),char(C)) :- write(C).
reducestring(Hd : Tl,Nhd : Ntl) :-
            Hd =>> Redhd , ! ,
            reducestring(Redhd,Nhd) ,
            Tl =>> Redtl , ! ,
            reducestring(Redtl,Ntl).

num(X) >>> num(X) :- write(X).

char(X) >>> char(X) :- write('%') , write(X).
```

```
logical(X) >>> logical(X) :- write(X).

X >>> fail :- is_failure(X) , write(fail).

nil >>> nil :- write('[]').

Hd:Tl >>> Nstr :-
            id(string) --> (Hd:Tl) =>> logical(true) , ! ,
            write('"') ,
            reducestring(Hd:Tl,Nstr) ,
            write('"').

Hd : Tl >>> Nhd : Ntl :-
              id(list) --> (Hd:Tl) =>> logical(true) , ! ,
              write('[') ,
              Hd =>> Redhd , ! ,
              Redhd >>> Nhd ,
              Tl =>> Redtl ,
              reducelist(Redtl,Ntl) ,
              write(']').

Hd : Tl >>> Nhd : Ntl :-
              Hd =>> Redhd , ! ,
              Redhd >>> Nhd ,
              write(':') ,
              Tl =>> Redtl ,
              Redtl >>> Ntl.

X >>> X :- write(X).
```

ABSTRACT

A VIRTUAL MACHINE TO IMPLEMENT PROLOG.

Gerard BALLIEU
Department of Computer Sciences
K.U.Leuven
Celestijnenlaan 200 A
B-3030 Heverlee (Belgium)

We describe the design and the definition of a vitual Prolog machine. Like other computers, this virtual machine has an instruction set and a working storage (sstatements and data).

The design of the instruction set is mainly based on the implementation by D. Warren on the DEC10 where he used an abtract machine to explain the principles involved in his compiler. The organization of the working storage corresponds to the "non-structure sharing" technique of C.S. Mellish or the "copying" approach of M. Bruynooghe.

One of our main purposes is of course to realise the idea of the virtual machine. The execution of a Prolog program on the vitual machine consists of two steps:

- Compilation of Prolog programs to virtual machine instructions. The compiler is written in Prolog and the compilation process should be completely reversible.

- Interpretation of the virtual machine instructions. An interpreter is being developed in a high level language (Pascal and C) and it should be mainly portable.

It is our goal to combine the advantages of both compiled Prolog (efficiency) and interpreted Prolog (adaptibility). We argue that this implementation is easily portable to different computer systems be rewriting only that part of the interpreter which implements the built-in procedures.

A VIRTUAL MACHINE TO IMPLEMENT PROLOG.

Gerard BALLIEU
Department of Computer Sciences
K.U.Leuven
Celestijnenlaan 200 A
B-3030 Heverlee (Belgium)

1. Introduction.

Prolog is a simple but powerful programming language based on symbolic logic. A lot of specific features such as declarative reading, incomplete data structures, unification and non determinism make Prolog programs very attractive and well suited for solving a great variety of problems. There is a growing interest to use Prolog as a software tool to design and develop new projects. In order to support Prolog as a real programming language, we design a Prolog system having the following charateristics:

- the Prolog system has to be efficient: compared with other languages the execution time must be reasonable (maximum 3 or 4 times slower) and the storage use may not overload the computer system.

- the Prolog system should be portable to a variety of machines and it should be easily adaptable to the specific capabilities of a particular computer.

- Prolog programs have to be compiled to virtual machine instructions which are completely machine independant.

- the data representation in the Prolog system should cover both Prolog implementations on conventional machines and on dedicated hardware.

In the next section we describe the main features (storage areas and instructions) of the virtual machine. Some design decisions are discussed and compared with the Prolog implementation of D. Warren [5]. Finally we discuss the current implementation and give some future developments.

2. Description of the virtual Prolog machine.

2.1. General processes.

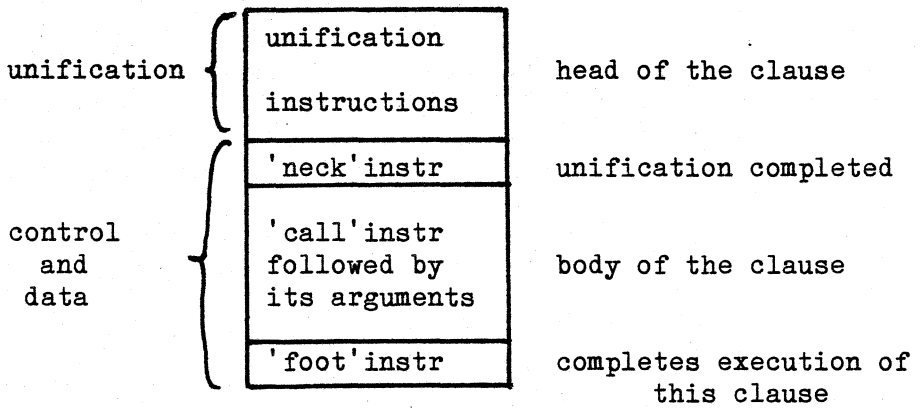
We design a virtual machine with an architecture which should support the efficient execution of Prolog programs. The execution mechanism of logic programs consists in constructing a sequence of

proof-trees according to the depth-first left-to-right search strategy [1] and to store in each node the appropriate variables and data. The fundamental questions we have to answer are of the form: "what does the machine do?" and "where and how does it represent its data?".

A Prolog machine has to perform two kind of processes: a control process and a unification process. On a sequential machine architecture these processes are alternated. The control process selects the next goal and the procedure definition, adjusts the proof-tree or restores the proof-tree to a previous state. The unification process is in fact a computation process which tests and assigns data or creates complex data structures.

To represent the control information and the data structures involved in the execution of a Prolog program, the virtual machine will provide a complex run-time structure consisting of an environmentstack, a copystack and a resetstack (or trail). For complex terms we use the "structure copying" approach.

The design of our virtual machine has strongly been influenced by the work of D. Warren [5] where he used an abstract machine to explain the Prolog compilation process. We also compile each Prolog clause into a sequence of virtual machine instructions according to the following scheme:



2.2. The main working storage.

The major data area of the virtual machine is the environmentstack. Like in block structured languages this stack is used to build a run-time environment for each goal (procedure call). When a new goal or subgoal is takled, a new stackframe is created and space is reserved for the variables and for linking (management) information.

The stack frames are linked in two kinds of lists: a father-list and an alternative list. Each stackframe belongs at least to one of the lists. The father-list corresponds to a path in the proof-tree from the root to the current node. The alternative list is a list of backtrackpoints or nodes with alternative choices to solve the goal corresponding to the node. In figure 1 we show for a given proof tree the corresponding environmentstack: P is the initial goal or problem, Di is a deterministic node and Bi is a backtrackpoint.

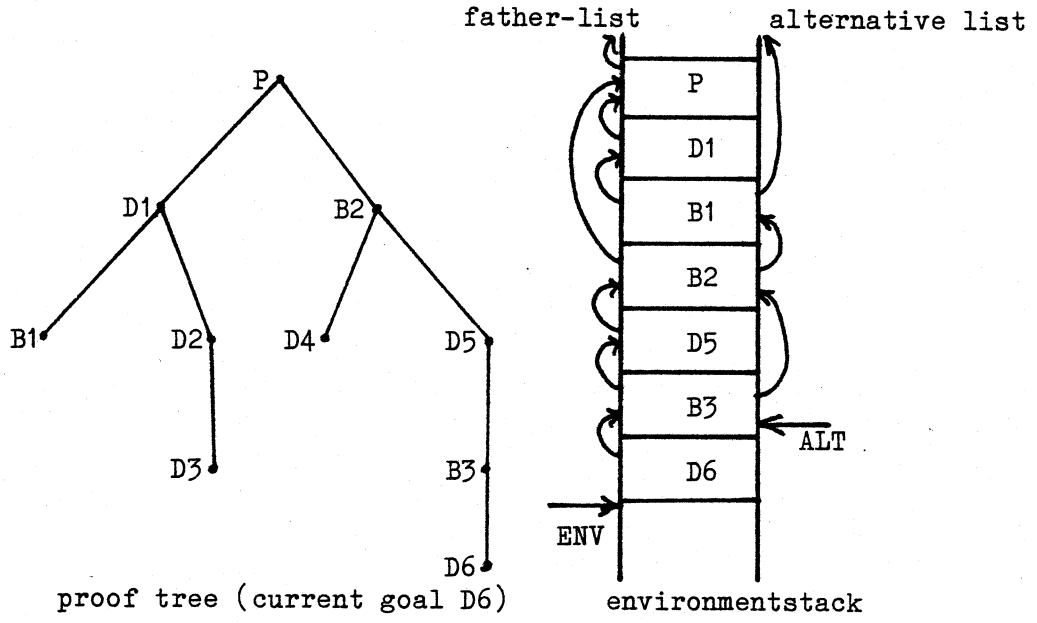


figure 1

The top element of the father-list and the alternative list is pointed by ENV respectively ALT. When a goal is successfully completed and no alternative choices remain (no backtrackpoint), the top frame of the stack (father-list) is removed. When a goal fails, the last backtrackpoint becomes the current frame and an alternative clause is chosen to solve the current goal.

Each stack frame also has space for the variables in the corresponding clause. Due to the general tree structures and the incomplete data structures in Prolog (dynamic data structures) it is not always possible to put the variable binding in the reserved space. When a variable's value is a constant (atom or integer) the value is put in the stack frame. When a variable is bound to a compound term (functor and arguments) a copy of this term is made and put on a second stack, the copystack, and a reference to this copy is put in the stack frame. Another reason for having two stacks is that on successful completion of a deterministic goal we will deallocate a stack frame and that for further computation we

still need the variable bindings. The value of a variable in the environmentstack can either be a constant, undefined (free variables), a reference to a compound term on the copystack or a reference to another variable earlier in the environmentstack.

The third working area of the virtual machine is the resetstack which is a trail or a push-down list. This area is used to store the addresses of variables which need to be reset to undefined (free) on backtracking.

The copystack and the resetstack generally increase in size with each new goal and are reduced by backtracking. The top elements are pointed by COPY respectively RESET and the old values of these pointers are kept in the mangement information part of the last backtrack frame. The management info contains also the links of the father-list and the alternative list, and pointers to the current goal and the alternative clauses if any.

Next to the working storage areas which are writable, we have the code area for storing the code of the compiled program. Information in the code area is generally accessed in a "read-only" manner.

2.3. The instruction set.

According to the control process and the unification process we can classify the virtual machine instructions in two classes: the unification instructions and the control instructions.

2.3.1. The unification instructions.

The main computation in Prolog consists of a sequence of unifications or pattern matching operations. Each unification involves matching two terms. One term is a "goal" (or procedure call) followed by its parameters and is instantiated. The other is the uninstantiated "head" of a clause. The control instructions verify that unification only takes place between a goal and a clause with the same name and arity. The unification process tries to match each of the arguments of the head of the clause against the corresponding arguments of the goal.

Instead of using a general matching procedure, the head of a clause is translated into unification instructions, most of which are simple tests and assignments. The arguments of the goal are translated into a sequence of literals (or "argument instructions").

The variables of a clause are categorised in three classes as follows:

- local variables: multiple occurences, with at least one in the body, numbered from 1 to n

- temporary variables: multiple occurences, all in the head of the clause, numbered from n+1 onwards

- void variables: single occurences.

The unification instructions are:

```
uvar(i)     : matching of the free variable i against ...
uref(i)     : matching of the bound variable i against ...
uint(j)     : matching of the integer value j against ...
uatom(a)    : matching of the atom a against ...
uvoid       : matching always succeeds
uterm(fn,n): matching of the functor fn with arity n against ...
```

(the number of a variable refers to a variable in the current frame.)

The literals (argument instructions) are:

```
var(i)       : the free variable i
ref(i)       : the bound variable i
atom(a)      : the atom a
void         : a void variable
funct(fn,n) : the functor fn with arity n
```

(the number of a variable refers to a variable in the goal frame.)

The next table gives an overview of the unification process:

goal / head	var	ref	atom	int	void	funct
uvar	assign	assign	assign	assign	assign	copy assign
uref	assign	general	case of	case of	success	case of general
uint	assign	case of	fail	test	success	fail
uatom	assign	case of	test	fail	success	fail
uvoid	assign	success	success	success	success	skip
uterm	copy assign	case of general	fail	fail	skip assign	test

```
assign : simple assignment
copy   : copy a compound term
test   : simple test (and assignment)
case of: multiple test
general: general unification algorithm
```

Most of the unification instructions are simple test and assignment instructions. If one of the terms is a reference we have to dereference that term until we get its value (undef, atom, int or funct). We can avoid long reference chains if we use only references to compound terms or to free variables. (Otherwise the value is copied.) The length of the reference chain would be mostly one.

There are two cases where we have to copy a compound term, depending on its source:

- the compound term appears in the haed of the clause

- the compound term appears in the argument list of the goal. Since the argument list is accessed in a read-only manner, only the parts containing variables must be copied. Therefore the compound terms are marked with "labelvar" or "labelcons".

There are three cases where the general unification algorithm can be invocated. This happens when two compound terms are to be unified and neither of them is known at compile time.

Remark that the virtual machine has no special instructions for initialising variables since the types "ref" and "var" indicate if a variable is free or bound.

2.3.2. Control instructions.

Each clause of a Prolog program is translated into a sequence of virtual machine instructions consisting of unification instructions for the haed of the clause, literals for the argument lists and control instructions (neck, call and foot).

- neck(n) : unification is completed; n is the number of local variables to be kept on the current environment.

- call(p) : this is a procedure call; a new frame is created, the call or return address is saved and a jump to address p is performed.

- foot : completes the execution of a goal, possibly removes the current frame and transfers control to the next instruction of the parent goal.

A Prolog procedure is composed of one or more clauses and is

translated into a list of control instructions of the form:

```
p : enter
    try(C1)
    try(C2)
      .
      .
    trylast(Cn)
```

enter : new procedure starts
try(Ci) : execute the instructions of clause Ci and note that there are alternative choices (backtrackpoint).
trylast(Cn) : execute the instructions of the clause Cn.

Note that these instructions manage the different clauses of a procedure and that they are generated at the end of the compilation process. If we extend our Prolog system with built-in predicates for adding or deleting clauses, this part of the code must be changeable.

Finally we have two control instructions which are strongly related to the Prolog source program: "cut" and "fail".

- cut(i) : i is the number of local variables; the alternative list must be adjusted and space can be recovered from the environmentstack.

- fail : forces backtracking.

2.4. Example.

As an example we show the quicksort program: source and virtual machine instructions.

```
qsort(.(X,L),R,R0):-partition(L,X,L1,L2),
                     qsort(L2,R1,R0),
                     qsort(L1,R,.(X,R1)).
qsort(nil,R,R).

partition(.(X,L),Y,.(X,L1),L2):-
lt(X,Y),!,partition(L,Y,L1,L2).
partition(.(X,L),Y,L1,.(X,L2)):-
partition(L,Y,L1,L2).
partition(nil,_,nil,nil).
```

```
3qsort1 : uterm(.,2)
          uvar(0)
          uvar(1)
          uvar(2)
          uvar(3)

3qsort2 : uatom(nil)
          uvar(0)
          uref(0)
          neck(0)
          foot
```

```
                neck(7)
                call(partition,4)
                ref(1)
                ref(0)
                var(4)
                var(5)
                call(qsort,3)
                ref(5)
                var(6)
                ref(3)
                call(qsort,3)
                ref(4)
                ref(2)
                labelvar(1)
                fn(.,2)                    3qsort : enter
                ref(0)                              try(3qsort1)
                ref(6)                              trylast(3qsort2)
                foot

4partition1 : uterm(.,2)        4partition2 : uterm(.,2)
              uvar(0)                         uvar(4)
              uvar(1)                         uvar(0)
              uvar(2)                         uvar(1)
              uterm(.,2)                      uvar(2)
              uref(0)                         uterm(.,2)
              uvar(3)                         uref(4)
              uvar(4)                         uvar(3)
              neck(5)                         neck(4)
              call(lt,2)                      call(partition,4)
              ref(0)                          ref(0)
              ref(2)                          ref(1)
              cut(5)                          ref(2)
              call(partition,4)               ref(3)
              ref(1)                          foot
              ref(2)
              ref(3)
              ref(4)
              foot

4partition3 : uatom(nil)          4partition : enter
              uvoid                            try(4partition1)
              uatom(nil)                       try(4partition2)
              uatom(nil)                       trylast(4partition3)
              neck(0)
              foot
```

3. Implementation.

As a first step in our Prolog system the Prolog source programs are compiled into a sequence of virtual machine instructions. A first version of the compiler has been written in Prolog itself. [6] The output consists of symbolic Prolog machine code as illustrated in the previous example and of two tables: a functor table (names of the predicates and arity) and an atom table.

The next step in our Prolog system is the interpretation of the virtual machine instructions. The interpreter should be query-oriented and has the following structure:

```
init read-only part (code area)
WHILE not end
  DO  read query
      compile query (set Program Counter to first instr.)
      init working storage
      execute (Program Counter)
      remove query
```

The initialisation part reads the symbolic code and transforms it into a sequence of word-codes which are loaded in the code area. The call instructions are divided in two classes: calls of evaluable predicates (built-in procedures) and calls of user-defined procedures. In the WHILE-loop a query is read and compiled into a sequence of word-codes which are added to the code area. This compilation can result in extending the atom table and the functor table. After execution of the query, the code area and the tables are restored.

In our prototype version we have split up the code in two parts:

- the executable part (unification and control instructions) is put in the code area

- the literal part (argument lists) is put on the copystack as a read-only segment. The literal part has the same structure as the compound terms except that a literal can be "var(j)" while a compound term on the copystack has the value "undef" instead of "var".

Figure 3 gives an overview of the Prolog system.

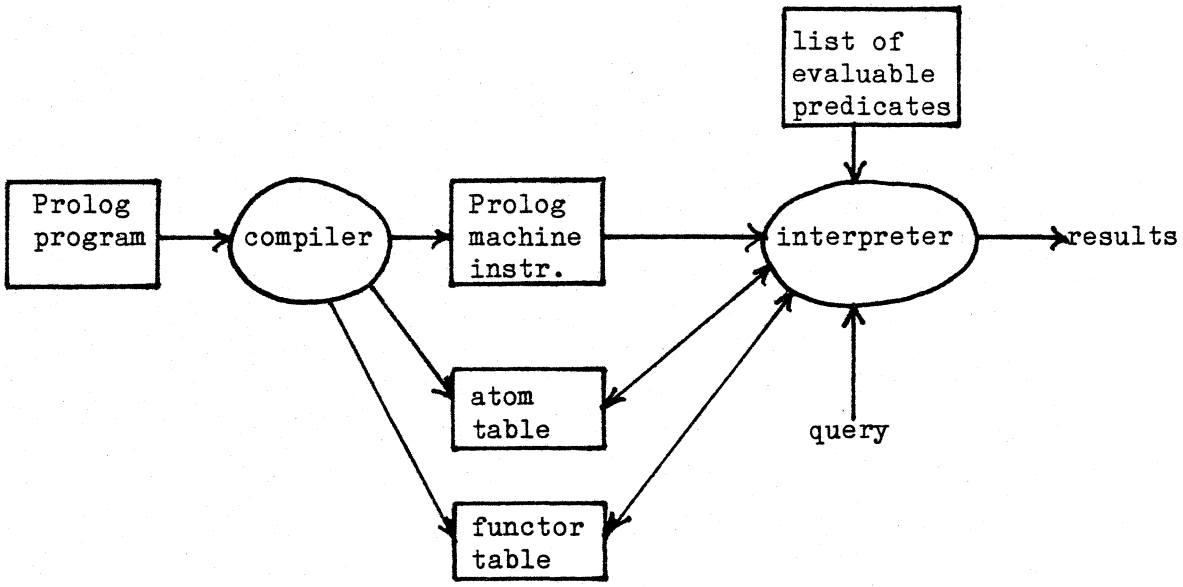


Figure 3

4. Design concepts.

Having described the main features of our Prolog system, we now comment some design concepts and their consequences.

- The "structure copying" approach is used as data representation technique for compound terms. Compared with the implementation of D. Warren, in our system there is no need to split up the variables in globals and locals: they are all local. A compound term is copied on the copystack only if it has variables. In addition the copying approach will behave better when a garbage collector is needed. [2]

- For the head of a clause we generate executable code for all terms nested to any level. We also detect the first occurrences of the variables in the body of the clause and the arguments of the goals are marked with "var" or "ref". Due to this decision we have eliminated the need to initialise the variables and the specific initialization instructions.

- The variables of the parent frame which are bound during the execution of a goal are never to be put on the resetstack because the arguments of the goal define which variables are free.

- The Prolog system has a modular strucure. Optimizations and extensions of the system require only small adjustments. The

implementation of the "neck"-instruction is responsible for tail recursion optimization. If we will add the "occurcheck" to the unification process, we only have to extend the implementation of "uref".

- The Prolog system is easily portable to other machines. If we will take full use of the capabilities offered by the underlying machine, it is sufficient to adapt the implementation of the evaluable predicates or to add new built-in procedures.

5. Future developments.

The virtual machine described in this paper is being implemented. A prototype of this machine has been written in the language C (under the UNIX operating system) and some simple Prolog programs have been tested. In comparison with the existing interpreter (written in C by M. Bruynooghe) our system behaves favourably in speed and space. For more complex programs we expect better results. Another implementation will be written in Pascal for machines with a Pascal-oriented architecture such as the PERQ.

We further plan to set up a complete Prolog program environment for this Prolog system:

- the current implementation will be optimized: tail recursion, clause selection based on the arguments, intelligent backtracking...
- development of a Prolog debugging tool
- different modules of a Prolog program may be compiled and linked into one executable program.
- the list of built-procedures and utility programs has to be extended
- the Prolog system has to be coupled with a relational database or with a database machine.

Acknowledgements

I am grateful to Maurice Bruynooghe for the many helpful discussions and to Gerda Janssens who has implemented and tested as a student project, this virtual machine in Prolog and C.

6. References.

[1] Bruynooghe, M. The memory management of PROLOG implementations, in Logic Programming (Clark & Tarnlund eds.), Academic Press, 1982.

[2] Bruynooghe, M. A note on garbage collection in Prolog interpreters, Proc of the first Int Logic Conf., Marseille, September 1982.

[3] Cloksin, W.F. and Mellish, C. Programming in Prolog, Springer Verlag, 1981.

[4] Mellish, C.S. An alternative to structure sharind in the implementation of a PROLOG-interpreter, in Logic Programming (Clark & Tarnlund eds.), Academic Press, 1982.

[5] Warren, D.H. Implementing PROLOG- Compiling Logic Programs, 1 and 2, D.A.I. Research Report No 39, 40, University of Edinburgh, 1977.

[6] Warren, D.H. Logic programming and Compiler Writing, Software Practice and Experience, Vol 10, nr 2, pp97-126.

The Personal Sequential Inference Machine (PSI):

Its Design Philosophy and Machine Architecture

Hiroshi Nishikawa Minoru Yokota
Akira Yamamoto Kazuo Taki Shunichi Uchida

Institute for New Generation Computer Technology
Mita-Kokusai Building, 21F.
4-28, Mita 1-chome, Minato-ku, Tokyo 108
Japan

ABSTRACT

As a software development tool of the Fifth Generation Computer Systems (FGCS) project, a personal sequential inference machine is now being developed. The machine is intended to be a workbench to produce a lot of software indispensable to our project. Its machine architecture is dedicated to effectively execute a logic programming language, named KL0, and is equipped with a large main memory, and devices for man-machine communication. We estimate its execution speed is about 20K to 30K LIPS. This paper presents the design objectives and the architectural features of the personal sequential inference machine.

1. Introduction

The final goal of the Fifth Generation Computer Systems (FGCS) project[1] is to develop the basic technology for a totally new computer system which has the ability to handle knowledge information. Inference is for the key mechanism in that system. That is the basic motivation why our project chose logic programming as the basic programming framework.

As one of the actual programming languages based on logic, there is an on-going active move of using Prolog to build new computer applications, especially in the artificial intelligence area. However, the processing power of existing computers is not sufficient for this purpose. It is quite important for the project to rapidly establish the logic programming environments. To satisfy this aim, the Sequential Inference Machine (SIM) is under the development in ICOT.

SIM is mainly intended to be a software development tool, however, the design of its architecture has many experimental aspects. It seemed to be difficult to design the ideal machine at once. So we have taken the following development steps:

1) designing a new programming language based on logic.

2) designing a personal sequential inference machine which is specialized for that new language.

3) designing a new operating system running on that new machine.

4) designing an advanced sequential inference machine based on the experiences from 1) - 3).

As the first step, the logic programming language, called Kernel Language Version 0 (KL0), was designed to take the place of Prolog. KL0 is mainly used to describe system software, such as the operating system kernel, compilers, and interpreters. Therefore KL0 can be regarded as a conventional assembly language except for logic programming features. And a Personal Sequential Inference machine (called PSI:) which is designed to execute KL0 is now under the development as the second step.

The following sections describe PSI design objectives, its system overview, and its machine architecture.

2. Design Objectives

As PSI is considered as a main computing tool in the initial stage of the FGCS project, the main requirements for its design are the high performance and an easy-to-use man-machine interface. Since PSI must be available as soon as possible, the main efforts are

focused on designing its processing unit and memory unit. However, in another aspect, designing PSI can be considered as an experimental step toward the target inference machine.

2.1 Performance Goal

As a software development tool, an adequate execution speed and a sufficient memory space must be provided in order to execute real-world applications.

On this point, it is suitable to compare them with the DEC-10 Prolog system[2]. Because, it is the most popular one and its compiler generates very fast codes.

However, the DEC-10 Prolog system is limited in its memory size (256K words) for users. It is relativity small for actual Prolog applications. This limitation may cause a serious problem in its use. The lower execution speed might be compensated by longer processing time, however, there is no way to continue the program execution if the system has used up such a memory space as a stack. From our experience in using the DEC-10 Prolog system, we estimated that at least a 10 times larger memory space must be necessary. In this situation, the virtual storage system would be an attractive feature, however, its implementation in the Prolog environment involves several problems to be studied. We have to study such problems more deeply as the swapping ratio between main memory and secondary storage, namely the locality of memory accesses, effective cache control mechanism, and an effective garbage collection algorithm working real-time[3]. Therefore we decided to leave the virtual storage system as a future extension. Instead of it, PSI is equipped with a relatively large real memory, maximum 16M words. About execution speed, PSI is designed to attain 20K to 30K LIPS (Logical Inference Per Second) which is the similar performance to the DEC-10 Prolog compiler version running on DEC-2060.

2.2 Personal Use

PSI is designed as a self-contained, personal machine in order to provide its user with powerful computing facilities and an efficient programming environment.

An easy-to-use, sophisticated man-machine interface is the most important features for software development tools. To provide good man-machine communications, PSI is equipped with a bit-mapped display device and a pointing device (a mouse). And a multi-window system is planned to be implemented on them. The input/output devices for Japanese characters will also be included, and PSI will support a word processing system for Japanese.

2.3 Local Area Network

PSI is planned to be connected to a local area network in order to give its user a more productive environment. Although any kind of peripheral devices can be connected to PSI, an usual PSI system will have a limited number of devices according to its own system characteristics.

Through a local area network, the distributed processing system connecting several PSI's can be built. Furthermore, the user can access other machines from PSI, such as a relational data base machine also being developed in the project, and conventional commercial machines.

2.4 Flexibility

PSI has adopted microprogrammed control for flexibility and extendability.

The project has decided on KL0 as a machine level language, however, its usefulness will be verified after PSI completes. In addition, the research and development of new programming languages, such as concurrent Prolog[4][5], is also one of the important subjects in the project. Therefore PSI must be able to execute those experimental languages as their test bed.

2.5 Evaluation

Using PSI, several items of measurements are planned for evaluation on programs behavior and machine design. One is to evaluate characteristics about the execution profile of logic based programs. Another one is to evaluate the validity about PSI architecture and hardware design. Especially, the measurement of memory access characteristics including cache hit ratio is one of the important items, because memory access is the most frequent operation in inference machines. The next advanced models of SIM will be designed utilizing effectively these evaluation results.

2.6 Specialized Hardware Supports

As a first experimental inference machine, an effort has been made to introduce several specialized hardware supports suitable for executing a logic programming language in PSI. To improve unification speed, PSI has hardware buffers. The role of these buffers is to quickly refer to the binding values of variables. For dynamic data type checking, each word has an 8 bit data tag (tag architecture). To

make memory access operations faster, the connection between the main memory and the processing unit was designed as tightly as possible.

PSI design objectives can be summarized as the combination of high performance of the 32 bit "super mini-computer" with the good man-machine interface of the "super personal computer".

3. System Overview

One of the key factors to determine a machine architecture may be the design of the machine instruction set. PSI is a specialized machine for executing the logic programming language (KL0),however, it must have its own operating system to be a self-contained personal machine. This section briefly summarizes the software system and hardware configuration of PSI.

3.1 Language System

One advantage of a logic programming language is to use its non-determinism effectively. However the non-determinate operation is considered unnecessary for describing low-level system control such as the kernel of operating systems, because it mainly consists of determinate operations and thus non-determinate operations would produce redundancy. In general, if a machine architecture is dedicated to some high- level programming language, it becomes difficult to implement its operating system in that language on the same machine. In this situation, a different programming language could be used for system description, however, this approach would degrade the uniformity of the system. And the machine architecture should support two different types of language processing. To make the entire system uniform, we decided to implement a PSI operating system based on the logic programming concept. KL0 has been designed to make this possible.

Figure 1 shows the language system hierarchy. The system programmer uses KL0 directly to develop a compiler, an interpreter, and operating system kernels. From the user's view point, KL0 can be regarded as a machine language of PSI, however, KL0 is basically a high-level, logic programming language. Its features are summarized as follows:

- o a subset of DEC-10 Prolog
- o an extended ability for hardware resource handling
- o an extended ability for interrupt handling and process control

o extended execution control facilities

KL0 includes the normal unification mechanism and clause handling mechanism like usual Prolog. From this view point the users can regard PSI as a complete Prolog machine. On the other hand, the users can also specify the machine level control with its extended facilities in KL0.

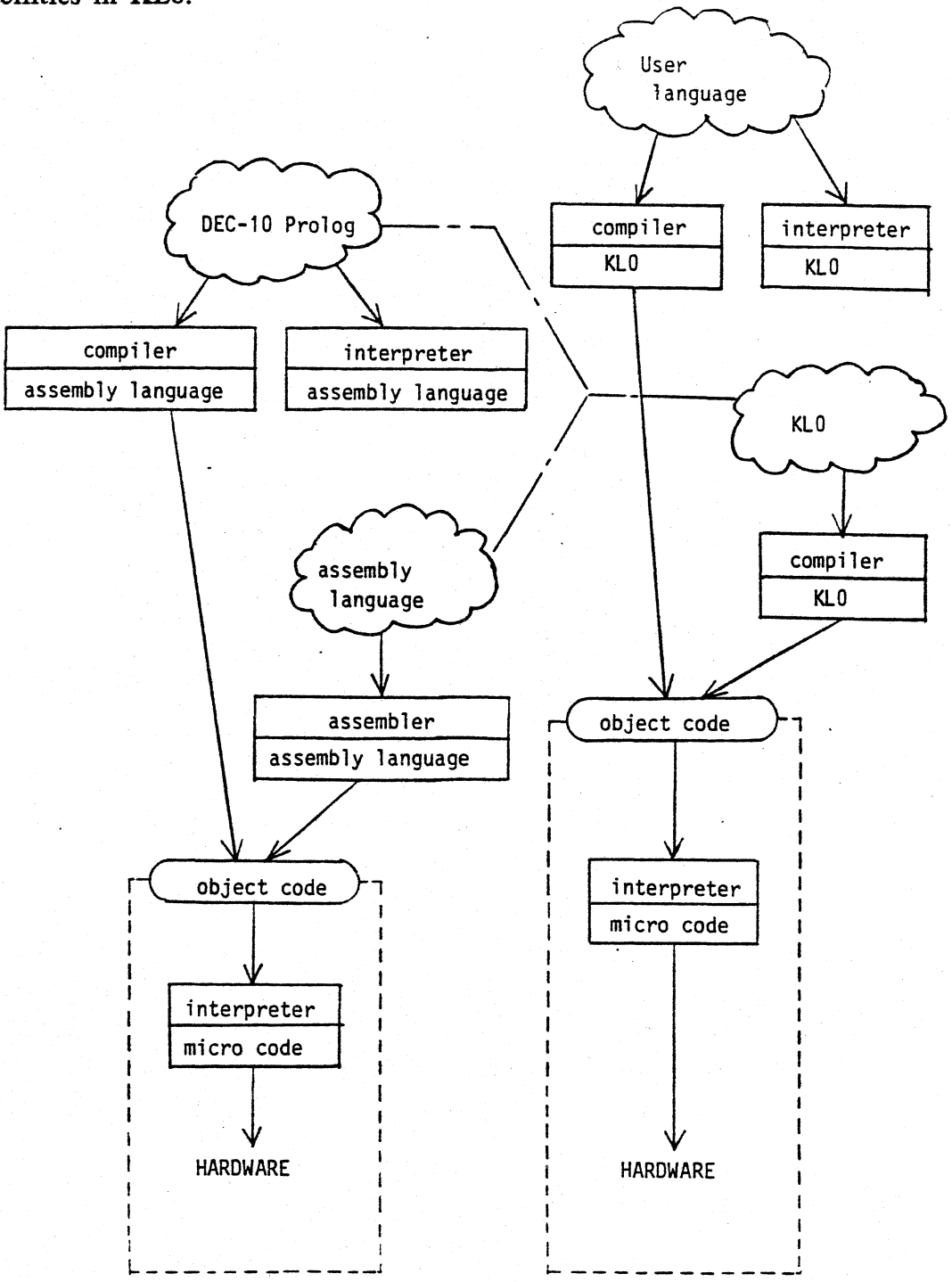


Figure 1. language hierarchy

3.2 Operating System Support

The kernel parts of the PSI operating system are written in KL0. These are transformed into internal machine forms by the compiler which is also written in KL0 itself. Then PSI hardware/firmware directly executes those internal forms. Furthermore, time crucial parts of the operating system kernel, such as the garbage collector or process switcher, are executed directly with firmware. The applications programmers will use a higher programming language than KL0. This language is executed with the interpreter or is compiled by the compiler written in KL0 into internal machine forms.

From the software side, it can be said that the PSI operating system is written in completely a logic programming style. From the hardware side, it can also be said that PSI architecture supports the primitive kernel operating system functions.

3.3 System Configuration

Figure 2 shows the PSI system configuration. CPU has a microprogram sequencer. The capacity of its writable control storage is 16K words. The micro instruction is 64 bit long and is executed in less than 200 nsec.

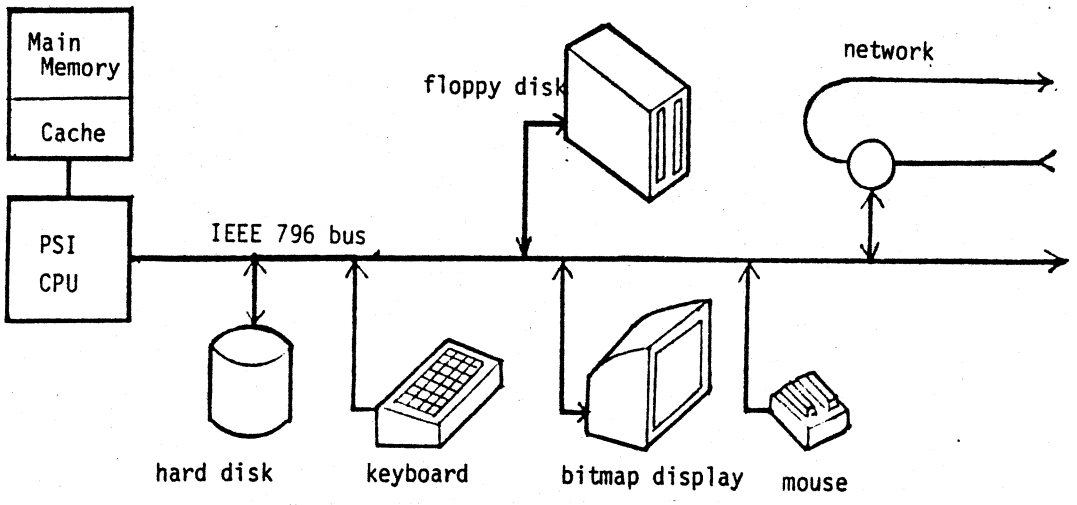


Figure 2. system configuration

CPU interprets internal object forms of KL0 with its micro-coded interpreter. Its hardware mechanism is mainly dedicated for the fast unification. It includes several discrete registers, register files, and an arithmetic operation unit.

The memory unit has a relatively large main memory instead of being equipped with a virtual memory system. A maximum of 16M 40 bit words can be installed. To shorten memory access time, PSI is equipped with a cache memory. It consists of 2 sets of 4K word memory, and a write-back strategy is adopted. Since several stack areas are required for interpretation of KL0 and each stack area will arbitrary grow during program execution, PSI introduces logical memory addressing. Therefore the roles of the memory control unit are address translation and cache control. If the required data exist in the cache memory, PSI can fetch that data within one micro instruction cycle.

A general purpose input/output bus is provided to PSI. To keep design simplicity and generality, IEEE-796 standard bus(MULTIBUS) is adopted. As a minimum configuration, PSI supports a fixed head disk, a floppy disk, a key board, a bit-mapped display, a mouse, a printer, and a local area network interface. Since PSI is planned to be connected to a local area network, the peripheral devices may be selected according to their own characteristics.

PSI also has an additional parallel interface port, in order to satisfy the requirement for connecting special I/O devices directly. For example,this parallel interface will be used to connect the relational data base machine or the voice recognition device, and etc.

4. Machine Architecture

The architecture of PSI was decided based on various considerations. A KL0 program is compiled into the internal object forms of PSI. But the level of the object code has been decided to be higher than that of ordinary machine instructions. So PSI is regarded as a high level language machine. In order to attain high performance, PSI adopted a tag architecture. Furthermore, a cache memory and special purpose registers are provided to improve the unification speed. PSI always refers to memory with logical address, and also has hardware supports for multi-processing.

4.1 How to Design the Machine Instruction Set

KL0 is a logic programming language, however, it is mainly used for system description. Therefore, performance in its execution is crucial. To take advantage of the source program information as much as possible, we decided to employ a compiler and thus PSI executes compiled codes instead of interpreting source codes directly. Even though, after compiling KL0, there still remains many operations to be performed only in execution time, such as unification, because of a

dynamic feature of the logic programming language. Then several levels of machine instructions can be considered.

The lowest one may be the conventional machine instruction level, and a KL0 program would be compiled into small pieces of those primitive machine instructions. The highest one is the internal form which is translated one by one from a source statement of KL0. The desirable machine instruction level depends on the characteristics of the language.

Originally, KL0 contains two different groups of elements. The first group is user-defined clauses to be executed within the logic programming framework. Namely, it is executed based on unification and backtracking. The execution of them is slightly simple and dominated with memory access operations. Therefore, it is undesirable that such execution is broken into many small machine instructions, because many instruction fetches are needed. In addition to this, there is less room for macro optimization on the hardware side because of low level machine instructions. This results in increased redundancies in both execution time and memory usage. We considered that PSI should have these unification and backtrack control facilities by itself.

The second group is built-in predicates for such operations as arithmetic operations and input/output operations[6]. Since the execution of them is performed determinately, their object codes can be represented in compact forms like conventional machine instructions. These built-in predicates are introduced not only to enhance the efficiency of frequently used operations but also to be able to include such primitive operations as register handling and direct memory manipulations used in the operating system.

Consequently the PSI machine instruction set has two types in its internal object forms. The first one corresponds to user-defined clauses. Actually, such a clause is compiled into the sequence of several internal object forms according to source clause definition. PSI interprets that sequence as a machine instruction on the whole. The others correspond to built-in predicates. Basically, a built-in predicate is compiled into one internal machine form.

4.2 Internal Object Forms

A KL0 program is translated into the corresponding internal object form described above. How to represent data and clause is shown in this section.

4.2.1 Internal Data Representation

Through examining PSI machine architecture, providing it with enough ability for increasing requirements from application areas is considered. At least 32 bits are necessary for representing sufficient magnitude of numbers and addressing space. In result, PSI employs a 40 bit word representation as shown in Figure 3. The upper 8 bits represent tag bits (tag part), and the remaining 32 bits represent the data itself (data part). 2 bits of the tag part are used by the garbage collector and the remaining 6 bits indicate the type of data included in the data part.

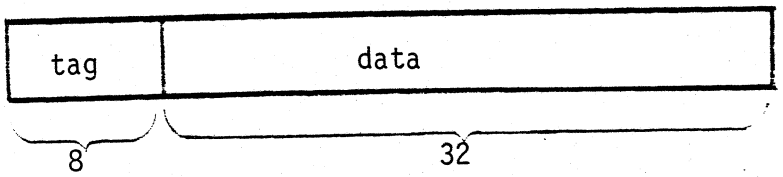


Figure 3. word format

PSI has several internal data types corresponding to ones in KL0. The visible data types for user are listed below:

- o symbol
- o integer
- o real
- o vector
- o string
- o local variable
- o global variable
- o void variable

(a) Symbol

This indicates the identifier of an atom. In the data part, the symbol number corresponding to an atom is stored. The printing image of an atom is managed by the operating system. So there is no direct relation between the symbol number and its printing characters.

(b) Integer, Real.

These are numerical data on PSI. The value of them is stored in the data part.

(c) Vector

A vector is a block of continuous memory slots, and is used to represent various structured data such as binary trees. As shown in Figure 4, a vector is usually accessed by way of its descriptor. However, this representation always needs an extra memory access whenever a vector is accessed. Since it is supposed that the vector which has a few elements is frequently used in programs, the direct vector type is introduced in order to effectively access such vectors. The conventional list structure is an example, and its representation

is shown in Figure 5. Comparing the performance of the structure sharing[7] with that of the copying strategy on structured data handling, PSI employs the structure sharing method similar to the DEC-10 Prolog. Therefore the structured data are manipulated as the pair of a structure (representing in a vector) and its values (located in the global stack). This address pair is called a molecule.

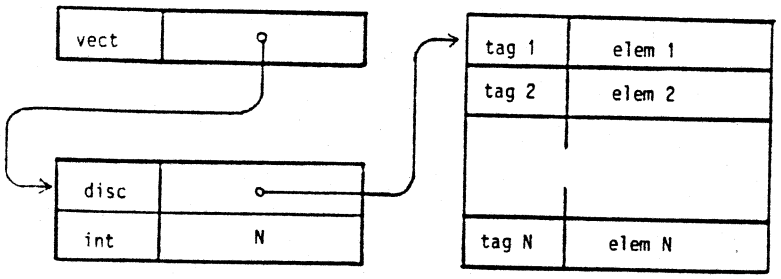


Figure 4. vector representation

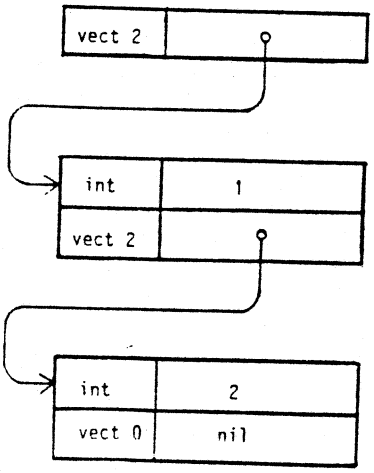


Figure 5. list representation

(d) String

A string data type is introduced for manipulating a byte (CHARACTER), double bytes (KANJI), and a bit (FIGURE) string data. Like the vector representation, string data is also accessed by way of its descriptor.

(e) Local/Global Variable

This data type indicates a local/global variable included in a clause. In the data part, the variable number is stored. The instance of a local variable is created in the local stack. The instance of a global variable is created in the global stack. Roughly speaking,the difference between the two is that the instances of local variables are cleared when the clause including them are executed

determinately, however, those of global variables are not cleared.

(f) Void Variable

This type means that the variable can have an arbitrary value, namely can be unified to any type of data.

4.2.2 Clause Representation

The definition of a clause in KL0 is the same as the one in Prolog. It consists of a head predicate and several goal predicates. The compiler translates a clause into a corresponding internal object form. As shown in Figure 6, each clause is represented as continuous memory slots, called code, in PSI. A code is a similar data type to a vector, and consists of a clause header, head arguments, goal predicate name and its arguments.

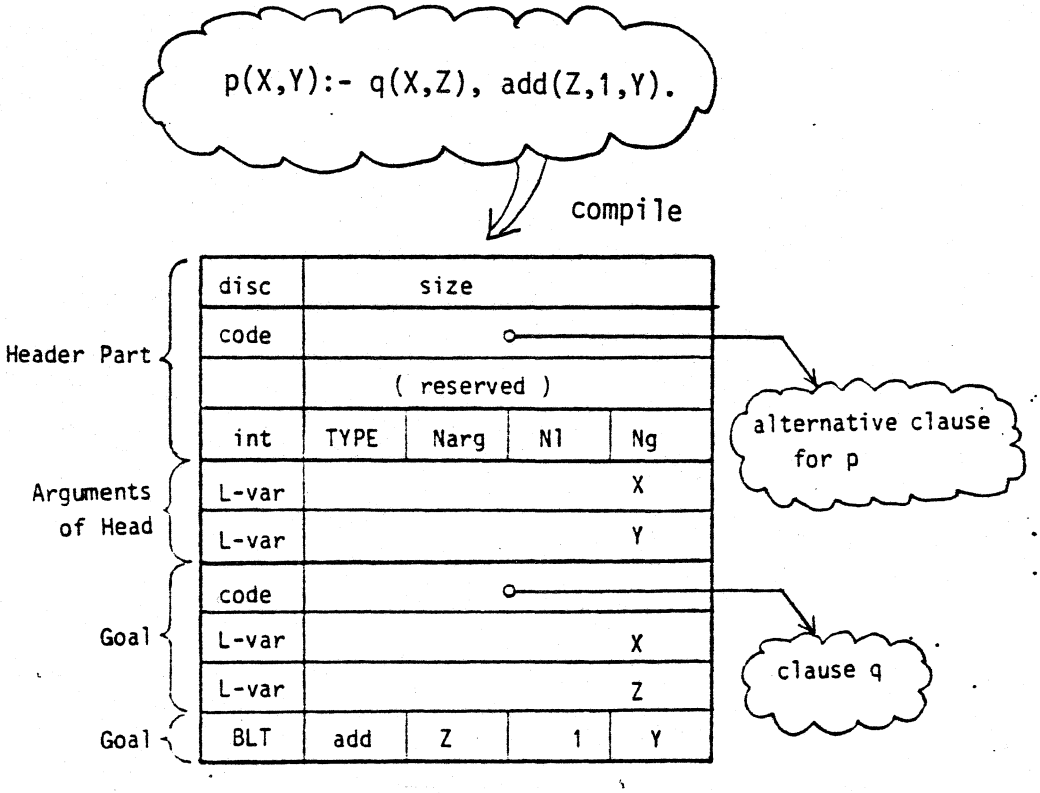


Figure 6. clause representation

A clause header consists of four words. The first word indicates the size of the code. The second word has an address to the code representing the next alternative clause. The third word is a reserved word. It might be used by the garbage collector. The last word indicates attributes of the clause. TYPE shows the clause type. For example, it is a unit clause, or having alternative clauses etc.

Narg shows the number of arguments included in the head predicate. Nl/Ng shows the number of local/global variables included in this clause.

Following a clause header, the head predicate arguments are located. Each argument is represented in the data types described in 4.2.1.

The remaining codes show the internal form of goals. There are two types of goal representation according as the called goal predicate is a built-in predicate or not.

(a) User-Defined Predicate Call

A goal predicate name is compiled into the pointer to the code representing the called clause. This pointer is stored in the data part and the tag of this pointer is set to a code type. The goal arguments are arranged continuously, following this pointer.

(b) Built-in Predicate Call

In the data part, a compact representation of machine instructions is stored and it consists of an 8 bit operation code and three 8 bit operands. The role of built-in predicates is to create objects, test the attributes of objects, and manipulate objects etc. A built-in predicates is compiled into one word object code basically, so that it can be executed efficiently on PSI.

Each goal is compiled into the pointer of the corresponding clause and its arguments. There are three connection types of goals, which PSI can directly interpret with its firmware interpreter.

(a) AND Connection

AND connection shows that the goals are combined as an AND node in the AND-OR search tree. Each goal is continuously located as shown in Figure 7-(a). AND connection means that each goal is executed sequentially and if a goal is failed, then backtracking occurs.

(b) OR Connection

This type is used to represent an OR connection included within a clause. OR connection shows that each goal is combined as an OR node in the AND-OR search tree. This connection is realized by an OR instruction as shown in Figure 7-(b). At first execution, the first goal is tried. When they fail, then the second goal is tried. Each branch of the OR connection can be composed of several goals. Therefore each branch of an OR connection is the same as an ordinary alternative clause except that they are included in only one clause and require no unification process.

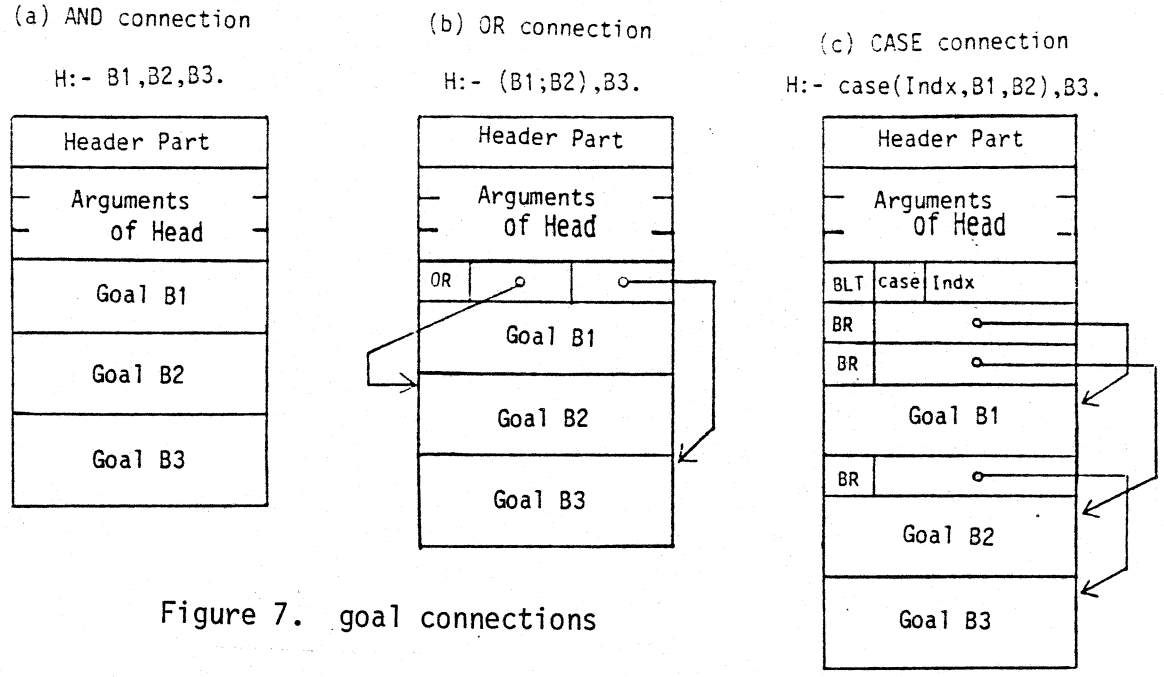


Figure 7. goal connections

(c) CASE Connection

CASE connection can be regarded as the arrangement of indexed goals. Figure 7-(c) shows the internal format of CASE connection. One of the goals is selected by the result of CASE instruction and if it is successively executed then the goal following the case block is executed next. Even if backtracking occurs, unlike OR connection, the remaining indexed goals are not executed.

4.3 Execution of PSI Internal Object Forms

For interpretation of the KL0 program, the following four stacks are needed:

- o local stack
- o global stack
- o trail stack
- o control stack

The use of these stacks is similar to those of DEC-10 Prolog, however, the control stack is separated from the local stack in order to efficiently execute the extra control primitives of KL0.

The local stack is an instance region for local variables. Preceding the unification process, PSI allocates the stack entries according to the number of local variables included in a clause. These stack entries are popped up when the evaluation of the clause including them is determinately terminated, or unification fails. They are also cleared when it is pruned by a "cut" operation.

The global stack is an instance region for global variables. Similar to the local stack, PSI allocates stack entries according to the number of global variables. These entries are only popped and cleared when unification fails. In addition, a molecule generated during unification and some control information are also allocated in this stack.

The trail stack is used for undoing variables when backtracking occurs. In this stack, binding information (i.e. the cell address where a value is stored during unification and whose content must be changed to 'undefined' when unificatoin fails) is stored. When the instance value of a variable is modified, its old contents are also stored in the trail stack in addition to its cell address.

In the control stack, various book-keeping information required for the execution control is stored. All of them are pointers which represent the execution environment of corresponding clauses. They are used to return to the calling clause, or to the backtrack point when unification fails.

There are some data types dynamically generated during program execution. Some of them are described below.

(a) Reference

It indicates a pointer generated during unification.

(b) Molecule

PSI adopts the structure sharing method to represent structured data described before. Since a molecule consists of two words in PSI, it can not be located into a variable cell. Therefore, a molecule itself is allocated in the global stack, and the reference to it is located in the variable cell.

4.4 Address space

PSI has a 32 bit logical address space. It is composed of 256 logically independent areas. The size of each area is 16M words, and managed by pages of 1K words. The reason why the concept of area is introduced is as follows:

(a) Since PSI supports multi-processing, it is desirable for the

operating system to assign completely independent areas to each process.

(b) Since PSI firmware interpreter uses four stacks described in section 4.3, it is desirable to be able to expand each stack area independently. If these stacks are allocated to the same space, a collision between stack areas will occur. At that time, one of them must be moved to another space. This situation causes serious overhead time.

Since four stacks are required for interpretation of a KL0 program, it means that each process needs at least four areas for its execution environment. On the other hand, code areas might be shared among many processes. If four areas are assumed to be used for code areas, namely heap areas, a maximum of 63 processes can be created on PSI from 256 areas.

Each area is divided into 16K pages. A page consists of 1K words. An area is managed by PSI operating system in page units. PSI allocates one page when a process needs more memory. On the other hand PSI disallocates some pages when a process release memory.

In result, the memory address field is divided into an 8 bit area number, 14 bit page number, and 10 bit offset as shown in Figure 8.

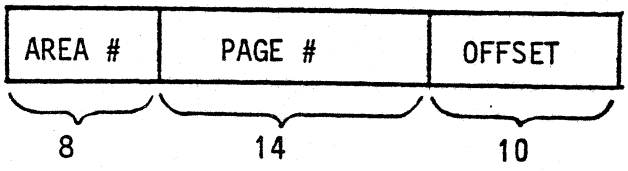


Figure 8. address format

The address translation mechanism is shown in Figure 9. The translation from a logical address to a physical address is performed with an area table and a page table. Each area table entry shows the base address of a page table located in the page map table corresponding to an area. And each page table entry shows the physical page address corresponding to a logical address page. As a first step to generate a physical address from a logical address, the area table is accessed using the area number, and a page table base address is obtained. Then the page map table is accessed using the sum of that page table base and a the page number. Finally concatenating the output of the page map table and the page offset, a 24 bit physical address is obtained.

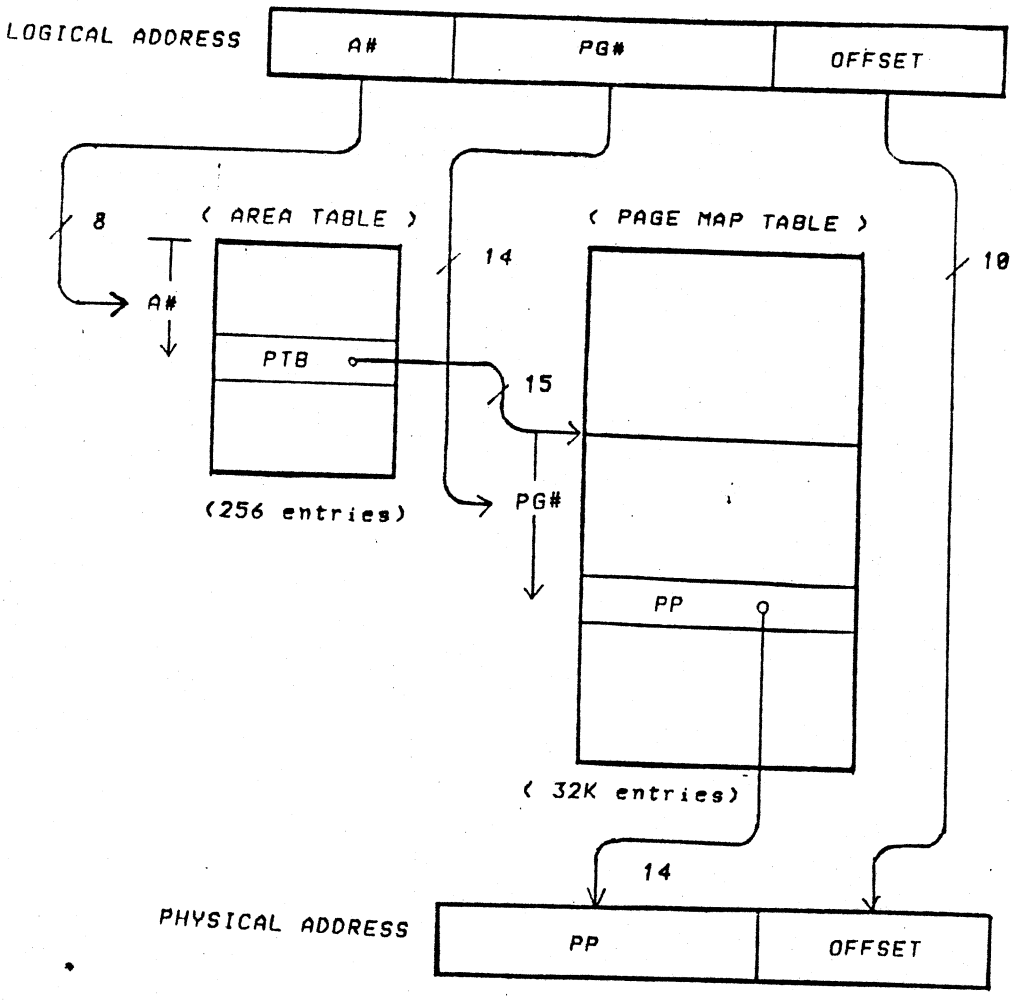


Figure 9. address translation

To achieve address translation, it is common to use a Translation Lookaside Buffer(TLB). Each TLB entry contains a logical page address and corresponding physical page address. TLB is a sort of cache memory, and if the address pair corresponding to a logical address is stored in it, there is no reference to the translation table existing in the main memory. PSI does not adopt this method. Instead, the area table and the page map table are located in special fast memory. In result, the address translation process is performed within a micro instruction cycle. The reasons why TLB is not adopted are shown below:

(a) If the address pair is not in TLB, the translation table in main memory must be accessed to generate a physical memory address.

(b) Since the garbage collector must search all memory space, it is supposed that the memory access locality during garbage collection is not so high. Therefore, TLB might not work well in that situation.

(c) PSI does not adopt virtual memory. The total amount of page map table entries can not exceed the number of physical pages. Since the maximum size of main memory is 16M words, it is sufficient to have 16K entries in the page map table.

The size of an area can extend from one page to 16K pages. Before program execution, the maximum number of pages used in a area cannot be predicted. Furthermore, a process needs at least four areas, however, the utilization of each area is different among processes. Accordingly, as the number of page table entries increases during execution, a page table may collide with another page table within the page map table. To avoid that case when possible, it is desirable to locate each page table corresponding to an area as dispersively as possible. Also, the page map memory size should be larger than the number of physical pages. To satisfy this condition, PSI has a page map memory of 32K entries. Since the standard physical memory size is 4M words, the size of a page map memory is eight times larger than that of physical pages.

If a page table collision occurs in page map memory, a trap occurs and page table relocation must be done. There are many algorithms to be considered. It is a future research theme to examine which algorithm is better.

4.5 Hardware Supports for Fast Unification

Unification plays an important role in executing a KL0 program. To efficiently execute the unification process, the hardware support mechanism is indispensable. The major part of the unification process is memory access and data type checking. The facilities employed in PSI are as follows:

- o Cache memory
- o Tag bits
- o Frame buffer

(a) Cache Memory

The merit of using cache memory is to reduce the cost of all memory accesses besides stack access. PSI adopts the write back-strategy for cache control, not the write-through strategy. A cache memory manages logical addresses. Therefore, if the accessed data exists in cache memory, no address translation is needed. The address translation is required only if a cache miss-hit occurs. PSI memory controller performs address translation during the cache memory access in parallel. This mechanism creates no overhead time for translation when the cache memory miss-hit occurs.

(b) Tag Bits

A tag is essential to effectively interpret a data type. To realize fast unification depends on how rapidly the data types can be examined. For this aim, tag bits are attached to all data, and they specify the type of the data. A special hardware mechanism, which decodes tag bit pattern efficiently, is provided in PSI.

(c) Frame Buffer

Frame buffer is the set of special registers provided for the top of the stack frame. In this buffer, the arguments of a clause and the cells of local variables are stored. Most of the unification is done using this buffer. This reduces the number of memory accesses, and faster unification will be realized. Furthermore, using this buffer, Tail Recursion Optimization (TRO)[8] can be realized efficiently.

4.6 OS Support

Since PSI is designed as a self-contained system, it requires own operating system. This operating system consists of an end-user interface (command interpreter), a programming system (editor, debugger), a file system, and so on. To provide its users with a sophisticated programming environment, that operating system must be an easy-to-use system, and provide good man-machine communications. Considering that these systems are specified by KL0,it is desirable that PSI must have operating system support functions.

To attain this objective, PSI has various hardware and firmware supports. For example, such primitive operations as a memory allocation or a garbage collection included in the memory management system is directly performed by firmware. The process switching of the process management system is also performed by firmware. Furthermore, PSI holds the process information in fast CPU memory in order to reduce process switching overheads. This is an essential hardware support in PSI, because KL0 requires larger execution environment than ordinary programming languages, and without that hardware support the contents of many base registers must be saved into the main memory at process switching.

In addition to higher level operating system support, there are several KL0 built-in predicates which perform low level system control, such as hardware resource handling, direct memory manipulation, and input/output control. These built-in predicates are effectively executed by firmware.

Besides this support described, garbage free regions is introduced to support the operating system kernels. In this region no garbage collection is done. This means that a program running in this region can be executed even while a garbage collection process is being executed. Those special processes unconcerned with the garbage collection are called supra GC processes. The aim of introducing this

GC-less process is to maintain good man-machine interface even when the garbage collector is working.

Summarizing those, the hierarchy from the end-user interface language to the hardware on PSI is shown in Figure 10.

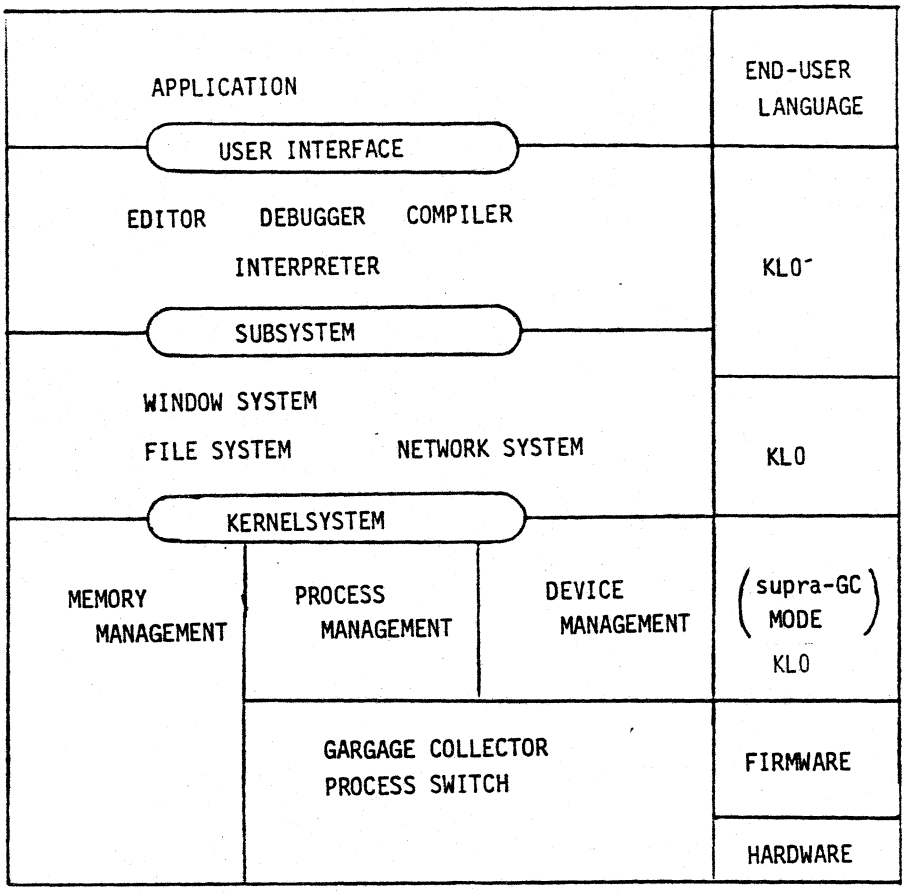


Figure 10. operationg system hierarchy

5. Conclusion

In this paper, we described the design objectives and the machine architecture of a Personal Sequential Inference machine, PSI. Its detail hardware design has almost been completed and the microprogrammed KL0 interpreter is now under the design. The rough estimation of PSI execution speed is comparable to the compiled codes of DEC-10 Prolog system on DEC-2060.

PSI is a first step toward the target inference machine which will be attained in ten years. For designing next advanced SIM, we are planning several evaluations on PSI. Many software products will also be made on PSI. We believe that PSI will be a powerful and useful workbench for our project.

ACKNOWLEDGMENTS

The authors express their grateful thanks to Dr. Takashi Chikayama for his valuable advice, and to Mr. Kazuhiro Fuchi, Director of ICOT Research Center and to Dr. Kunio Murakami, Chief of First Research Laboratory for their continuous encouragement, and to other members of ICOT for their useful comments and discussions.

REFERENCE

[1] Outline of Research and Developments for Fifth Generation Computer Systems. ICOT Research Center, April (1983)

[2] Warren, D.H.D. Implementing PROLOG - compiling predicate logic program. Vol.1-2, D.A.I Research Report No.39-40, Department of Artificial Intelligence, Univ. of Edinburgh (1977)

[3] Cohen, J. Garbage Collection of Linked List Data Structures. Computing Surveys, 13-3 (1981)

[4] Shapiro, E.Y. A Subset of Concurrent Prolog and Its Interpreter. ICOT Technical Report TR-003(1983)

[5] Takeuchi, A., et al. Interprocess Communication in Concurrent Prolog. Logic Programming Workshop, '83 (1983)

[6] Chikayama, T., et al. Fifth Generation Kernel Language. Proc. of the Logic Programming Conference '83 (1983)

[7] Boyer, R.S and J.S.Moore. The Sharing of Structure in Theorem Proving Programs. Machine Intelligence Vol.1-7, Edinburgh Up (1972)

[8] Warren, D.H.D. An Improved PROLOG Implementation Which Optimizes Tail Recursion. D.A.I Research Report No.141, Department of Artificial Intelligence, Univ. of Edinburgh (1980)

A PORTABLE PROLOG COMPILER

D.L. Bowen, L.M. Byrd

Dept of Artificial Intelligence
University of Edinburgh

and W.F. Clocksin

St Cross College, Oxford

ABSTRACT

This paper describes the basis of the design of a Prolog implementation which is currently being built. This new implementation is intended to combine a high degree of portability with speed and efficient utilisation of memory. Our approach is to compile Prolog clauses into instructions for a relatively high-level abstract machine. This abstract machine is implemented by an interpreter written in a high-level systems programming language (C), giving a portable Prolog system.

Some portability must be sacrificed, however, in order to achieve the high speed required. The design is well suited to tailoring for particular machines, because there is a small central core of the interpreter which does most of the work. This central core can be translated into assembly language or microcode when necessary.

An advantage of this approach is that it avoids the compiler/interpreter dichotomy found in DEC-10 Prolog and LISP systems with compilers. All clauses are compiled, but compilation is reversible so that it is not necessary to have a separate representation of the textual form of clauses.

1. Introduction

This paper describes some design principles behind current work at Oxford and Edinburgh Universities to build a new Prolog system. The desired qualities of the new system are that:

(1) It should be highly portable.

(2) It should be fast and use memory efficiently; this requirement directly conflicts with (1).

The approach we have chosen is to compile Prolog clauses into code for a relatively high-level (i.e. Prolog oriented) abstract machine. This abstract machine is implemented by an interpreter written in a high-level systems programming language (C). The compiler, and many of the evaluable predicates, are written in Prolog itself. This approach has allowed us to get a preliminary version of the system running fairly quickly.

However, this system as it stands will not meet our requirement for speed. A certain amount of non-portable work will be necessary in order to achieve high speed on particular computers. Our intended methodology is to translate the most heavily used parts of the C code into assembly code, or microcode where possible (e.g. on the ICL Perq). This non-portable work is minimised because the central core of the interpreter is simpler and smaller than that of a direct Prolog interpreter.

We have opted for the sructure-copying method of [Mellish 80] and [Bruynooghe 80], rather than structure-sharing [Warren 77]. An important reason for this is that structure-copying is expected to give better locality of reference and therefore better paging behaviour on virtual memory systems. Another advantage is that it allows us to dispense with holding the Prolog form of all the clauses in the heap: our abstract machine is so arranged that we can reconstruct these terms when they are needed (i.e. in the implementation of the evaluable predicates 'clause' and 'retract') by effectively decompiling the compiled form of the clauses.

Our storage management strategy is basically that of [Warren 77], i.e. there is a heap containing the program, a "local" stack for control information and variable bindings, a "global" stack for structures, and a "trail" stack which keeps track of when variables are bound so that they can be reset to "uninstantiated" at the appropriate time on backtracking. One change is that a reference count is maintained for each clause so that pointers to clauses (as returned by the predicate clause/3 in DEC-10 Prolog) can safely be included in asserted terms. A consequence of this slightly complex memory management is that it is never necessary for a garbage collector to do a full sweep of the heap; it only has to sweep the local and global stacks.

As our run-time system is based on previously published work [Warren 77] [Warren 80], we will concentrate in the rest of this paper on the new part of our design which is the intermediate language.

2. The Intermediate Language

In this section we introduce the kernel of the intermediate language into which Prolog clauses are translated. Although this language subset has only seven instructions, it is sufficient; the only reason for extending it is for efficiency as will be discussed later. We introduce it syntactically by discussion of the (reversible) compilation of a Prolog clause. The semantics of the language will be explicated in the following section by means of a simple interpreter for it written in Prolog.

A compiled clause has two main parts: an External Reference (XR) table, and a block of byte-codes. Let us consider the compilation of the clause:

```
p(tp1,tp2,...)  :- q(tq1,tq2,...), r(tr1,tr2,...).
```

where the tpi, tqi and tri are arbitrary terms. The general form of the byte-code block is then:

```
<code for tp1>
<code for tp2>
 ...
enter
<code for tq1>
<code for tq2>
 ...
call <XR offset for procedure q>
<code for tr1>
<code for tr2>
 ...
call <XR offset for procedure r>
exit
```

This introduces the three "control" instructions we need: 'enter', 'call' and 'exit'. The 'enter' instruction simply marks the division between the head and the body of the clause. Each 'call' has an argument (the next byte-code in the block) which refers to an entry in the XR table which is a reference to the required procedure. Finally, 'exit' marks the end of the clause.

The terms which are the arguments of the head of a clause, and those which are the arguments of goals, are all translated in the same way. Each term is compiled into "data" instructions as follows:

(1) If the term is atomic it is translated as

```
const <XR offset>
```

where the corresponding entry in the XR table is either an integer (if the term is an integer) or a pointer to an atom record.

(2) If the term is a variable it is translated as

```
var <number>
```

where the variables in the clause are numbered in order of appearance.

(3) If the term is compound it is translated as

```
functor <XR offset>
<code for 1st argument>
<code for 2nd argument>
 ...
pop
```

The 'functor' instruction refers to an XR table entry which points to the corresponding functor record. It is followed by the compiled form of each of its arguments, followed by a 'pop' instruction.

For the purposes of the interpreter to be presented in the next section, we need to represent compiled code as Prolog data structures. Compiled procedures will be represented as assertions of the form:

```
procedure( Name/Arity, List_of_Clauses ).
```

A clause will be represented by a term:

```
clause( XR_Table, Number_of_Variables, List_of_Bytecodes )
```

An XR table is also represented as a term:

```
xrtable(...)
```

where the table entries are either integers, atoms, functors (written in the form Name/Arity), or procedures (written as procedure(Name/Arity)).

For example, the compiled form of the procedure:

```
append(nil,L,L).
append(cons(X,L1),L2,cons(X,L3)) :- append(L1,L2,L3).
```

looks like this:

```
procedure( append/3, [
    clause( xrtable(nil), 1,
        [ const, 1,                                  % nil
          var, 1,                                    % L
          var, 1,                                    % L
          exit]),
    clause( xrtable(cons/2,procedure(append/3)), 4,
        [ functor, 1, var, 1, var, 2, pop,           % cons(X,L1)
          var, 3,                                    % L2
          functor, 1, var, 1, var, 4, pop,           % cons(X,L3)
          enter,
          var, 2, var, 3, var, 4, call, 2,           % append(L1,L2,L3)
          exit ])]).
```

3. An Interpreter for the Intermediate Language

We now present our mini-interpreter written in DEC-10 Prolog. For simplicity, we use the unification and backtracking capabilities of Prolog rather than doing everything explicitly as is necessary in a real implementation. A consequence of this is that cut cannot easily be implemented in the mini-interpreter.

The entry point to the interpreter is the procedure arrive/3. Its arguments are the procedure to be called, a list of its arguments, and a continuation list which represents goals still to be solved. E.g. to append one list to another we would call:

```
:- arrive(append/3,[cons(a,cons(b,nil)), cons(c,nil), L],[]).
```

This call should succeed, instantiating L to :

```
cons(a,cons(b,cons(c,nil))).
```

There are two clauses for arrive/3 (Figure 1). The first of these finds any compiled clauses for the procedure. It then non-determinately selects (using member/2) the first clause, i.e. future failure will cause us to backtrack here and select another clause if there is one. Next it creates a new set of (uninstantiated) variables by means of the built-in predicate functor/3 which sets Vars to be the functor with name 'vars' and having Nvars uninstantiated arguments. Finally control is passed to execute/6 to execute the byte-code list (which we have called PC because it corresponds to the Program Counter in a real implementation).

The second clause for arrive/3 allows the built-in predicates of Prolog to be used in the mini-interpreter.

```
arrive(Proc,Args,Cont) :-
        procedure(Proc,Clauses), !,              % Find clause list for Proc
        member(clause(XR,Nvars,PC),Clauses),     % Select one
        functor(Vars,vars,Nvars),                % Make new set of variables
        execute(PC,XR,Vars,Cont,Args,[]).        % Go to execute byte-codes
arrive(Name/Arity,Args,Cont) :-
        Proc =.. [Name|Args],                    % No compiled clauses: call
        call(Proc),                              %   normal Prolog procedure
        execute([exit],_,_,Cont,_,_).            % and continue

member(X,[X|_]).
member(X,[_|L]) :- member(X,L).
```

Figure 1: arrive/3

The clauses for execute/6 (Figure 2) are all determinate, so that it resembles a CASE statement in other languages. Let us consider the data instructions first, assuming for now that they are in the head of a clause (i.e. before the 'enter' instruction).

The 'const' instruction is fairly straightforward: it simply matches the first element of the argument list with the appropriate entry in the XR table. (arg(X,XR,Arg) unifies Arg with the Xth argument of the term XR.) If successful, it then tail-recursively calls execute/6 to execute the subsequent instructions with the rest of the argument list. Note that if Arg was initially uninstantiated it will have become instantiated to the given constant. Similarly, 'var' matches the given variable with the current argument.

For 'functor' we first check that the argument has the right principal functor (or instantiate it to the most general term with this principal functor if it is uninstantiated). If successful, we obtain the list Args of the arguments of Arg and go to execute subsequent instructions which are to be matched against them. There remains the list Arest of arguments to be matched after Arg. This list is stacked on Astack from where it is

```
execute([const,X|PC],XR,Vars,Cont,[Arg|Arest],Astack) :- !,
        arg(X,XR,Arg),                          % Match XR entry with Arg
        execute(PC,XR,Vars,Cont,Arest,Astack).
execute([var,V|PC],XR,Vars,Cont,[Arg|Arest],Astack) :- !,
        arg(V,Vars,Arg),                        % Match variable with Arg
        execute(PC,XR,Vars,Cont,Arest,Astack).
execute([functor,X|PC],XR,Vars,Cont,[Arg|Arest],Astack) :- !,
        arg(X,XR,Fatom/Farity),                 % Get functor from XR table
        functor(Arg,Faton,Farity),              % Match principal functors
        Arg =.. [Fatom|Args],                   % Get Args of Arg term
        execute(PC,XR,Vars,Cont,Args,[Arest|Astack]).
execute([pop|PC],XR,Vars,Cont,[],[Args|Astack]) :- !, % Pop Args off Astack
        execute(PC,XR,Vars,Cont,Args,Astack).
execute([enter|PC],XR,Vars,Cont,[],[]) :- !,
        execute(PC,XR,Vars,Cont,Args,Args).     % Initialise diff list:
execute([call,X|PC],XR,Vars,Cont,[],Args) :- !,
        arg(X,XR,procedure(Proc)),              % Extract proc name from XR
        arrive(Proc,Args,[frame(PC,XR,Vars)|Cont]).   % Save context & go
execute([exit],_,_,[frame(PC,XR,Vars)|Cont],[],[]) :- !,
        execute(PC,XR,Vars,Cont,Args,Args).     % Resume previous context
execute([exit],_,_,[],[],[]) :- !.              % No previous context: stop
```

Figure 2: execute/6

removed by the corresponding 'pop' instruction.

We have explained how the data instructions work in the head of a clause. It is the 'enter' instruction that ensures that they also work in the body, where what they are required to do is build up rather than take apart the argument list. What it does is initialise a difference list: a partially formed argument list is the difference between the 6th and 5th arguments of execute/6. For example, if two arguments have been processed we would get a goal of the form:

```
:- execute(_,_,_,_,X,[<arg 1>,<arg 2>|X]).
```

Thus each data instruction encountered in the body appends an argument onto this argument list by instantiating the variable at the end of it to [<argument>|<new variable>]. It is interesting to see how this works for 'functor': this is left as an exercise for the reader!

The 'call' instruction terminates the difference list by instantiating the variable at the end to []. It then goes off to arrive at the called procedure with the new argument list, first stacking all the information needed to resume this clause on the continuation list.

Of the two clauses for 'exit', the first is selected when the continuation list is non-empty. It causes resumption of a clause after the successful completion of a 'call'. Note that it is necessary to reinitialise the difference list here so that another argument list is constructed for the next 'call'. The second clause for 'exit' terminates the program.

4. Some Additions to the Intermediate Language

It may be noticed that there is no point in returning from the last 'call' in a clause and restoring its context only to immediately 'exit' and restore a previous context. This can be avoided by introducing a new 'depart' instruction which replaces the last 'call' and the subsequent 'exit' (cf. [Warren 80]). The interpreter is easily extended to handle this new instruction by the addition of one more clause for execute/6:

```
execute([depart,X],XR,Vars,Cont,[],Args) :- !,
        arg(X,XR,procedure(Proc)),
        arrive(Proc,Args,Cont).
```

This is just like 'call' except that no continuation frame is stacked.

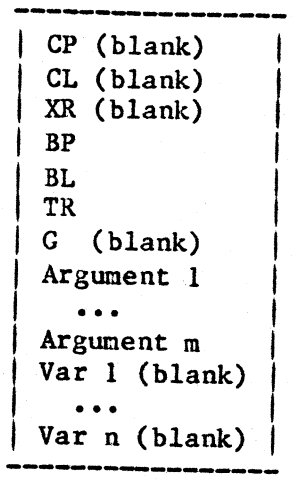
Another inefficiency arises in the execution of 'functor' if it appears as the last argument in the clause head, or as the last argument of some other term. In either case there are no remaining arguments (Arest is [] or will be instantiated to [] later) but we are stacking Arest anyway and popping it back to no useful purpose when 'pop' is encountered. The cure is to introduce another new instruction, 'lastfunctor', which is like functor except that it has no corresponding 'pop'. It is interpreted thus:

```
execute([lastfunctor,X|PC],XR,Vars,Cont,[Arg],Astack) :- !,
        arg(X,XR,Fatom/Farity),
        functor(Arg,Fatom,Farity),
        Arg =.. [Fatom|Args],
        execute(PC,XR,Vars,Cont,Args,Astack).
```

Various other instructions can be introduced to save space in the clause representation or to gain speed. An example is <'immediate' N> which allows a small integer N to be represented directly in the byte-code block without the need for an XR table entry. It is also useful to provide instructions for the simpler built-in predicates such as integer/1, var/1 etc. A possibility is to combine some of the instructions with their most common arguments to make new single-byte versions of two-byte instructions, but the trade-off with increasing the size of the interpreter needs to be studied empirically.

5. Considerations for a Practical Implementation

The operation of our environment (or local) stack, which holds continuation and backtrack information as well as the arguments of procedures and variable bindings, is based closely on [Warren 80]. At the point where we are about to commence execution of a byte-code block, the top frame of this stack is like this:



Continuation (byte-code) Pointer
Continuation Local stack frame
XR table for continuation
Backtrack Point (clause pointer)
Backtrack Local frame
Trail marker
Global stack marker

The first three words of the frame (marked blank because they have not yet been filled in) are for exactly the continuation information that was in the continuation stack of the mini-interpreter: the CL pointer allows access to the variables of the contination frame. The next four words are for control of backtracking. Then come the arguments to the procedure, which have already been filled in, followed by the variables which have not.

An argument register, A, is initially set to point to Argument 1. Each byte-coded instruction matches against the argument pointed to by A and then increments it. When a 'functor' instruction matches against an uninstantiated argument, it creates a new term with uninstantiated arguments on the global stack, and A is then set to point to the first of these new arguments. The previous value of A is saved on a special stack so that it can be retrieved by the corresponding 'pop'.

We do not actually have to initialise all the variables in the local stack frame to be "uninstantiated". The first occurrence of <'var' N> in a clause (for each N) is changed to be a new instruction <'firstvar' N> which simply assigns the value indicated by A to variable N. If a variable only appears once in a clause, there is no point in doing even this much work, so there is also a 'void' instruction which does nothing.

Another improvement we can make is to overlap the variable and argument blocks in the stack frame. That is, if a variable appears at the top level in the head of a clause, e.g. L2 in append([X|L1],L2,[X|L3]) :- ...), then we can use the appropriate argument slot for the variable value, thus saving space and avoiding superfluous assignments. All that has to be done is rearrange variable frame offsets appropriately (variables are not actually numbered 1,..,n, but by their offsets in the frame), and use 'void' instead of 'firstvar'.

Without special-purpose hardware, there is bound to be inefficiency in the way we have described building terms: first we build the term with all its arguments uninstantiated, and then subsequent instructions match against these uninstantiated arguments and fill them in. This involves (unnecessary) testing to see if each argument is uninstantiated; also it is in general necessary when instantiating a variable to test whether or not it

should be put on the trail. We avoid all this checking, and the need for initialising the arguments of the constructed term, by introducing a new mode of interpretation of our instruction set. This is called 'copy' mode, as opposed to 'match' mode which is what we have been discussing until now. In 'copy' mode data instructions simply copy the data they stand for over to A.

This concept of interpreter modes can also be useful for debugging. In normal operation, the abstract machine goes to great lengths to throw away any information which it will not need again. When debugging, this is undesirable, so we plan to include a 'debug' mode in which more information is kept.

One other complication should be mentioned. This is the problem described in [Warren 80] of dangling references arising from tail-recursion optimisation. We follow his approach of putting variables which may give rise to this problem onto the global stack. For this purpose we require two new instructions which are global stack versions of 'var' and 'firstvar'.

6. Related Work

A compiler for Prolog has been written in POP11 by C.S. Mellish at Sussex University. This actually compiles Prolog into the POP11 abstract machine language which is then in turn compiled into real machine language. Advantages of this approach are (1) relative ease of implementation, and (2) instant access to a good programming environment. The long-term drawback, however, is that there is no possibility of tailoring the memory management to the special needs of Prolog. The fully general POP11 garbage collector has to be used (even for backtracking).

Another approach has been taken by [McCabe 83]. His Abstract Prolog Machine is specified at a much lower level than ours, and depends on the availability of a LISP style garbage collector of some sort.

7. Conclusions

The design we have described is a compromise between pure interpretation and pure compilation. Preliminary tests have shown our initial system to be comparable in speed with Pereira's C-Prolog interpreter [Pereira 82]. It has the advantage over pure interpretation that it is easier to optimise for particular hardware: the kernel of the interpreter is relatively simple and compact and well suited to microcoding.

Our design requires much less space for program storage than pure compilation, due to the relatively high level of the byte-code instructions, and to the fact that we do not need to store a separate representation of the Prolog source code. Also there is the advantage that there is no dichotomy between interpreted code (that you can debug) and compiled code (which goes fast) as there is on the DEC-10 system. Finally, our design has the advantage of minimising the amount of machine-specific work which needs to be done in implementation.

It is our belief that people are going to want to run larger and larger ("knowledge-based") programs, and that therefore the efficiency of both program storage and garbage collection will become increasingly important. Prolog does not require the generality of a LISP or POP garbage collector, so it should have an advantage over these languages if more efficient, special-purpose memory management is used.

8. Acknowledgement

We are indebted to David Warren and Fernando Pereira for the inspiration behind this work.

References

[Bowen 82] D.L. Bowen (ed.), L.M. Byrd, F.C.N. Pereira, L.M. Pereira and D.H.D Warren,
"DECsystem-10 Prolog User's Manual",
Department of Artificial Intelligence,
University of Edinburgh, 1982.

[Bruynooghe 80] M. Bruynooghe,
"The Memory Management of Prolog Implementations",
In "Logic Programming", ed. K.L. Clark and S.-A. Tarnlund,
Academic Press, 1982.

[McCabe 83] F.G. McCabe,
"Abstract Prolog Machine - A Specification",
Technical Report, Department of Computing,
Imperial College, London, February 1983.

[Mellish 80] C.S. Mellish,
"An Alternative to Structure Sharing in the
Implementation of a Prolog Interpreter",
In "Logic Programming", ed. K.L. Clark and S.-A. Tarnlund,
Academic Press, 1982.

[Pereira 82] F.C.N. Pereira,
"C-Prolog User's Manual, Version 1.1",
Edinburgh Computer Aided Architectural Design,
University of Edinburgh, September 1982.

[Warren 77] D.H.D. Warren,
"Implementing Prolog - Compiling Logic Programs",
Research Reports 39 & 40, Department of Artificial
Intelligence, University of Edinburgh, 1977.

[Warren 80] D.H.D. Warren,
"An Improved Prolog Implementation which Optimises Tail
Recursion",
Proceedings of the Logic Programming Workshop, Debrecen,
Hungary, ed. S.-A Tarnlund, July, 1980,
(also available as Research Paper 141, Department of
Artificial Intelligence, University of Edinburgh).

84

Methodology of Logic Programming

by

Ehud Shapiro

Department of Applied Mathematics
The Weizmann Institute of Science
Rehovot 76100, ISRAEL

In this session I would like to discuss research methodology, rather than programming methodology, of logic programming. As a basis for discussion I propose the following statements, interspersed with text that attempts to justify or explain them.

1 Goals of research in logic programming

Statement: *Logic-programming share the goals of computer science at large.*

As I see it, there are no major differences between the goals of computer science at large and the goals of logic-programming. Both want to solve the problem:

(*) How to make computers do what we want them to?

This problem has two derivatives:

(1) How to make it easy for us to make computers do what we want them to?
(2) How to make computers do fast what we want them to?

Using a more respectable jargon, the two derivative questions become:

(1') How to program computers?
(2') How to make computers run the programs fast?

Many of us beleive that logic-programming may provide better solutions to these problems than the more conventional approaches to computer science.

Statement: *The basic method of computer science is bootstrapping.*

Computer science offers one encompassing methodology for solving these two questions, namely bootstrapping. Apart from brilliant new ideas (which no methodology can promise to provide), the crucial factor dictating the ease in which we can program computers and build faster computers is the computerized support available to these tasks. More concretely, the ease of programming is determined mostly by the quality of the programming environment available; and the possibility of building cheaper and faster computers is determined mostly by the quality of the CAD/CAM systems available.

Logic programming has a lot to contribute to the general thrust of bootstrapping, and also to provide some briliant new ideas of its own.

2 Prolog

Statement: *Prolog is an expressive and efficient programming language, which we are still learning how to use.*

Prolog is the first practical logic programming language. It is acquiring a growing group of users, who develop for it a rich set of programming idioms and techniques, and a refined programming style. Inspite of the initial dissatisfaction of Prolog's inventor, Alain Colmerauer, and others with the language, it turns out that its expressiveness is far greater than what was expected. It is surprising that such a simple language can lend itself to so many sophisticated and powerful programming techniques. Even after programming in Prolog for the past three years, Iam still learning new methods and techniques of Prolog programming.

Prolog falls short of the aspirations of the founders of logic programming in several respects: it has a rather inflexible control, and, to implement substantial systems, must resort to features that have only procedural meaning (cut, I/O, side-effects). Nevertheless, many of us beleive that Prolog, as it is, is a good programming language for many applications. To increase its effectiveness, Prolog requires improvements in its speed of execution and programming environment.

Statement: ***Prolog should be made to run faster.***

The desire for greater speed needs no justification. If we had as many MegaLIPS as we have MIPS, then almost no programming task will have to be carried in a lower programming language.

The speed of Prolog on a von Neumann machine can be increased in several ways. One is to improve the basic cycle of Prolog, the unification, by providing faster (cached, pipelined, multiported) memory access, by supporting the basic unification instructions in hardware or microcode, and by parallelizing the unification of subterms. Another is to incorporate in Prolog an abstract data-types mechanism, that will support in a logical way interface to efficient data-structures. Abstract data-types are needed in order to make more efficient use of the resources of the underlying hardware. If substantial systems are to be implemented in Prolog, then, given current technology, we cannot afford to represent everything as a list of elements. Mutable arrays, strings, and other efficient data-structures need to be supported. The only clean way to support these in the logic-programming framework is via predicates over abstract data-types.

Improving Prolog's speed by incorporating high-level parallelism (in contrast to the low-level parallelism available in the unification algorithm), is discussed in the next section.

Statement: ***Prolog needs a better programming environment. Prolog programming environments are best implemented in Prolog.***

Given the short time they exist, and the number of man-years devoted to their development, some of the current Prolog programming environments are quite impressive. The main problem that prevents further, or faster, development, is that every new Prolog implementor reinvents the wheel. This phenomenon is most evident in the development of the Edinburgh Prolog family, in which every new implementator has implemented the programming environment from scratch.

One of the most important properties of Prolog is that it is an excellent language for developing its own environment. By defining a small core-Prolog, which is expressive enough to implement a full environment, a new Prolog implementor can simply implement this core, and port the environment from a previous implementation. The availability of such a core will also support distributed implementation efforts, in which different tools are implemented in different locations. This is in contrast to the current situation, in which the burdon of implementing a reasonable environment for Prolog falls solely on the implementor of the core Prolog and its close associates.

My experience suggests that a subset of the system predicates of Waterloo Prolog, or those of Edinburgh Prolog augmented with the 'retry' and 'ancestor-cut' predicates, are expressive enough to implement almost any tool desired. Many system predicates in Edinburgh Prolog are better viewed as utilities that are, in principle, implementable in core-Prolog, but are provided for the sake of convenience or efficiency.

Statement: ***Prolog should be kept a small.***

There are two good reasons for keeping the core of Prolog small. One is intellectual economy. I think we are still learning how to program in Prolog. A baroque set of features (cf. IC-Prolog[4]) will prevent us from identifying what is essential and what is superfluous, and will not encourage the development of innovative programming techniques that squeeze every ounce of expressiveness from a small set of constructs.

Another good reason to keep Prolog small is related to the discussion of programming environments and bootstrapping. If there is a small Prolog core, in which everything else is implemented, then:

1. Developing a new or better Prolog requires less effort.
2. A new Prolog implementation does not need a new environment.
3. Sophisticated Prolog programming tools, that know about all core-Prolog system predicates, are easier to develop.

An example of an enhacement to Prolog that can be implemented in core-Prolog is a module and type system. Most current Prolog implementations resemble an assembly language, rather than a high-level programming language, in their flat name space of procedures, and in their lack of support of any type system. It is clear that a facility for modular programming is necessary for substantial systems to be developed in Prolog by many programmers. MProlog [14] supports a notion of modules. However, its implementation is in a low-level language, and cannot be ported to other Prologs. On the other hand, [5] showed that modules can be implemented easily in Prolog, by preprocessing, without affecting the Prolog core. Since the preprocessor is written in Prolog, it can be ported, in principle, to any compatible Prolog implementation.

Another example is the Prolog-10 debugger [1]. It is implemented almost solely in Prolog, but pieces of it which are implemented in a lower level language (pseudo-Prolog), prevent it from being easily ported to a new, compatible, Prolog implementation, such as CProlog. On the other hand, the debugging algorithms in [12] are implemented solely in Prolog.

Inspite of what is said, investigating extensions to Prolog is still a useful activity. I beleive that any extension to Prolog's core should satisfy at least the following three criteria:

1. It can be demonstrated, with non-toy examples, that the extension is useful.
2. The extension cannot be implemented in Prolog (e.g by preprocessing).
3. The extension can be implemented efficiently, and does not incure runtime overhead when not used.

Examples of extensions that can be implemented in Prolog are second order predicates (setof, bagof) [15] and modules [5]. Examples of extensions whose implementation seem to induce run-time overhead even when not used are selective backtracking [10] and several other forms of more sophisticated control [4], [11].

In addition to supporting its own environment, I beleive that Prolog is, in principle, an ideal language for implementing a VLSI CAD system. The main requirements of such a system are the ability to integrate a large database with algorithms that manipulate it. No other programming language supports both database and algorithmic functions in the way Prolog does. The main obstacle to realize such a pratical system today seems to be Prolog's inefficiency.

3 Concurrency

Statement: ***Prolog is not suitable for expressing concurrency.***

Inspite of its expressivness in general, Prolog has a major blind-spot: it is not suitable for expressing concurrency. This means that in order to build a Prolog machine we must either extend Prolog substantially, or use a lower programming language to implement multi-tasking. Needless to say, multi-tasking is an essential feature even in sequential computers.

There have been many proposals to incincorpoe more sophisticated control-constructs to Prolog, to support coroutinning and concurrency, e.g. in IC-Prolog, Prolog-II, MU-Prolog, and Epilog, among others. It seems that none of those is both expressive and efficient enough to implement a multi-tasking operating system.

Statement: ***Concurrency and don't-know nondeterminism (deep backtracking) do not mix well, but can be interfaced.***

The Relational Language of Clark and Gregory [2] and Concurrent Prolog [13] take a different approach. They give up Prolog's non-determinism (implemented by deep backtracking) for the sake of expressing concurrency. The memory management of these languages is very different from that of Prolog, therefore integrating the two efficiently on a von Neumann machine is a non-trivial problem. Also, my experience with programming in Concurrent Prolog suggests that applications that require concurrency do not require non-determinism, and vice versa. A good interface between Prolog (or a logic programming-based database machine) and concurrent logic-programming languages are set expressions, or Prolog's setof predicate, as suggested by Clark and Gregory [3]. The availability and sufficiency of such an interface reduces the need for an immediate integration of the two languages.

4 Parallelism

Statement: ***Parallel execution of Prolog is difficult.***

Logic programs offer two kinds of parallelism: Or-parallelism and And-parallelism. Or-paralelism means trying several candidate clauses in parallel. And-parallelism means trying to solve several goals in a conjunction in parallel.

One approach to designing a logic programming language for parallel computers is to patch Prolog. However, since Prolog was designed specifically

for efficient execution on a von Neumann machine, it is not clear that it is a good starting point. Adding Or-parallelism to Prolog is not so difficult concenptually. One problem is the cut. If cut is used to implement implicit negation (a substitute for if-then-else) and defaults, then Prolog programs may behave incorrectly when executed in Or-parallel mode. This is a difficult problem, due to the pervasiveness of this use of cut. Another problem is memory management. The overhead of maintaining seperate environments for the Or-parallel subcomputations may eliminate the benefits gained from their parallel execution.

Incoprorating And-parallelism into Prolog is far more difficult.

Statement: ***And-parallelism and Or-parallelism have different applications, and are best explored independently.***

Since the problems of parallel computers are so difficult, and there is so little positive experience with them in other branches of computer science, I think it is much more sensible to start small.

The first step is to examine the uses of the two kinds of parallelism. Or-parallelism is useful for speeding the solution of problems that require search. One significant class of search problems are database queries. In many applications, however, good algorithms can often provide a substitute for simple brute-force search. And-parallelism is useful for implementing parallel algorithms. The class of problems for which efficient parallel algorithms have been designed is increasing rapidly. The existence of computers that can actually run them will no doubt increase the pace in which they are produced. These observations suggest that And- and Or-parallelism have different, disjoint applications, which, at least initially, are best studied seperately.

The design of an Or-parallel database machine is an important and challenging problem. The close relationship between logic programs and relational databases suggests that ideas and concepts from relational databases can readily be put into use within the logic programming framework.

Concerning And-parallel machines, my view is that simple languages such as the Relational Language and Concurent Prolog are a good starting point. These languages are expressive enough in their current form for a parallel implementation of them to be useful and interesting. Hence their incporporation with the more powerful features of sequential Prolog may be postponed until the problems of building a parallel machine for these simpler languages are better understood.

5 Logic vs. control

91

Statement: *Efficient algorithms cannot always be obtained by twiddling with the control of logic programs.*

Some of the research on logic programming was guided by the desire to find some 'philosopher's stone': a notation that eliminate the need to think algorithmically. Kowalski's celebrated equation [6]:

$$\text{Algorithm} = \text{logic} + \text{control}$$

have suggested to many [4],[7],[8],[10], that if only we could find the 'right' control regime, we could factor the task of devising and implementing efficient algorithms into two, independent subtasks: defining the logic of a solution to a problem, and converting it into an efficient algorithm by imposing control on it. I beleive that this interpretation of the equation is too strict. It is impossible, in general, to specify sophisticated algorithms just by modifying the control component of a logic program. No massaging will make a logic program that specifies the exponential generate-and-test permutation sort into quicksort. The same statement is certainly true for less basic algorithms.

Statement: *Sophisticated control has large runtime overhead, hence it is best implemented in an embedded language.*

The sophisticated control regimes developed in response to Kowalski's equation usually have an unacceptable runtime overhead. Hence they cannot be incorporated in a base language. An alternative way to achieve sophisticated control is to implement embedded languages in Prolog. Implementing interpreters for embedded languages in Prolog is by now a well understood technique [9].

Statement: *Compile-time optimizations are superior to runtime optimizations.*

One of the goals of sophisticated control is to make certain logic programs run faster. This approach may be called "runtime optimization".

Whenever a runtime optimization of an inefficient logic program represents a tracktable algorithm, it is usually possible to implemente the algorithm directly in ordinary Prolog. The transformation of inefficient logic programs to efficient Prolog programs may be called "compile-time optimization".

It is my (unsupported) beleif that compile-time optimizations represent a more promising approach than runtime optimizations.

6 The Fifth Generation Project

Statement: *Logic-programming machines will require new solutions to old problems.*

One goal of the Fifth Generation project is to construct computers with a new machine language, based on logic. To realize such machines we will have to address many questions which are already solved for von Neumann computers. There is a lot to learn from the old solutions, but one measure for the viability of logic-programming is the quality of the new solutions it will provide to these old problems.

7 References

[1] Lawrence Byrd, *Prolog-10 Debugging Facilities,* Technical Note, Department of Artificial Intelligence, Edinburgh University, 1980.

[2] Keith Clark and Steven Gregory, A Relational Language for Parallel Programming, In *Proceeedings of the ACM Conference on Fucntional Programming Languages and Computer Architechture,* pp. 171–178, 1982.

[3] Keith Clark and Steven Gregory, *PARLOG: A Parallel Logic Programming Language* (Draft), Technical Report DOC 83/5, March 1983.

[4] Keith Clark and F. McCabe, The Control Facilities of IC-PROLOG, In:*Expert Systems in the Micro Electronic Age,* D. Mitchie (ed.), Edinburgh University Press, pp. 122–149, 1981.

[5] Paul Egghart, Logic enhancement. In *Proceedings of the ACM Conference on Lisp and Functional Programming Languages,* August, 1982.

[6] Robert A. Kowalski, Algorithm = Logic + Control, *CACM* **22(7)**:426–346, July 1979.

[7] John McCarthy, *Coloring Maps and the Kowalski Doctrine,* Technical Report STAN-CS-82-903, Stanford University, April 1982.

[8] Lee Naish, *An Introduction to MU-Prolog,* Technical Report 82/2, Department of Computer Science, University of Melbourne, 1982.

[9] Luis M. Pereira, Logic Control with Logic, In *Proceedings of the First International Logic Programming Conference,* pp. 9-18, ADDP, Marseille, September 1982.

[10] Luis M. Pereira and Antonio Porto, Selective Backtracking, In *Logic Programming,* K.Clark and S.-A. Tarnlund (Eds.), Academic Press, 1982.

[11] Antonio Porto, Epilog: a language for extended programming in logic, In *Proceedings of the First International Logic Programming Conference* pp. 31-37, ADDP, Marseille, September 1982.

[12] Ehud Y. Shapiro, *Algorithmic Program Debugging,* ACM Distinguished Dissertation Series, MIT Press, 1983.

[13] Ehud Y. Shapiro, *A Subset of Concurrent Prolog and its Interpreter,* Technical Report TR-003, ICOT — Institute for New Generation Compputer Technologu, 1983.

[14] SZKI. *MProlog Language reference Manual,* SZKI, Budapest, Hungary, November 1982.

[15] David H. D. Warren, Higher order extensions to Prolog — are they needed? In *Machine Intelligence* **10,** D. Michie, J. Hayes, and Y. H. Pao (eds.), Ellis-Horwood, 1982.

The pragmatics of Prolog: some comments

by

E.W. Elcock

The University of Western Ontario

London, Canada

N6A 5B9

Abstract

Logic programming and Prolog in particular have done much to illuminate the relationship between logic and computing. The relationship between logical consequence and effective construction is, however, very subtle, and it seems appropriate (even if not entirely novel) to continue to be concerned about too simplistic an approach to the difficulties which face any assertative programming language. This note attempts to focus some of these difficulties for Prolog in the context of a simple but rewarding pedagogic example.

(Keywords: Logic programming; Prolog; programming methodology, pragmatics)

Introduction: "between the expectation and the reality lies the shadow"

The paper comments on the view that Prolog, as an exemplar of logic programming, is a candidate for a specification language and as such provides specifications with a declarative (standard model theoretic) reading, but with the bonus that such specifications can be re-interpreted procedurally and without change as providing implementations of the specifications.

As a specification, a Prolog program [A,G] is to be thought of a sequent A => G. It is well-known, however, that Prolog is an incomplete system: that is, there exist Prolog programs [A,G] where A is a sequence of Horn clauses and G a conjunction of predications such that the corresponding clausal sequent A => G is a true sequent and yet the Prolog program [A,G] does not terminate successfully. A simple example illustrating this incompleteness is the Prolog program [A,G] where A is the sequence of clauses

1. mem(U,[V|L]) :- mem(U,L)

2. mem(U,[U|L])

and G is the goal statement

mem(a,[a|M])

In executing the goal statement Prolog repeatedly uses clause 1 in the procedure for list membership generating the infinite sequence of goal statements

mem(a,[a|M])
mem(a,M)
mem(a,M1) where M is bound to [V1|M1]
mem(a,M2) where M1 is bound to [V2|M2]

etc.

Prolog does not prove the true sequent A => mem(a,[a|M]). In fact, with 'mem' specified with the ordered pair of clauses above in some more general sequence of clauses A , Prolog will not establish the truth of any sequent involving a call of 'mem' on a list with variable tail.

In this example a simple reordering of the clauses of A would result in acceptable computational behaviour. Indeed, the set of Prolog proofs generated with the reordering is a superset of those generated with the original ordering. In this particular example, with A the sequence of clauses

1. mem(U,[U|L])

2. mem(U,[V|L]) :- mem(U,L)

the membership relation is very well behaved and acts synthetically (constructively) as well as analytically. Thus, if G is a goal such as

mem(a,[b|M])

then the Prolog program succeeds with, for example, M bound to [a|M'].

Prolog programmers might rationalize the problem of which this example is a symptom by insisting that, although one wishes to take advantage of the model theoretic semantics of Horn clauses in viewing a Prolog program as a lucid specification, one should be willing in Prolog as in any other language, to rewrite (transform) one's specification, now viewed as a program, with the pragmatics of the procedural interpretation (defined by the particular interpreter or whatever) in mind.

In the particular case of 'mem', and with the procedural interpretation of Prolog firmly in mind, one might rationalize away any unease with some argument that "it's obvious that one should have given the base case of the recursion first" and in this Prolog is no worse than a "conventional" applicative language where the "same thing" might have happened. Certainly Prolog programmers are well aware of the problem and acknowledge it (see for example Clocksin & Mellish, 1982). My readings, however, lead me to believe that many still do not treat the phenomenon with the seriousness it deserves. This note exploits one of my own five-finger exercises using Prolog in the hope of drawing further attention to the problem.

97

Permutations of a problem

Let's now turn to a more focal example: that of using Prolog to write a specification of a solution of the eight queens problem. We specify a board position by an ordered pair of row and column numbers (r,c) 1=<r,c=<8 . We wish to specify the set of subsets of size 8 such that no two members of a subset lie on the same row, column or diagonal. We take advantage of the fact that the interpretation of a sequence of the eight numbers 1 to 8 as a set of ordered pairs (r,c) , where c is the r'th number in the sequence. guarantees that no two members of the set have the same row or column number. With this representation of subsets we simply have to restrict the sequences to be such that no two (r,c) pairs in the represented subset are on the same diagonal.

With this preamble we might begin to specify a solution to the eight queens problem with the Horn clause

```
1.  queens(Q) :- perm([1,2,3,4,5,6,7,8],Q) , dsafe(Q)
```

with the intended interpretation that perm(M,N) specifies that the lists M and N stand in the (symmetrical) relation permutation to one another, and that the predicate 'dsafe' will be suitably specified to capture the intended interpretation discussed above.

So far, so good: the specification has a very clear model theoretic (declarative) semantics contributing to our intended interpretation - still to be filled out by specifications of 'perm' and 'dsafe'. Let's look at the following specification taken from Clark and McCabe's treatment of the eight-queens problem (1979):

```
1.  perm([],[])

2.  perm(L,[U|M]) :- inserted(U,L,L1) , perm(L1,M)

3.  inserted(U,[U|L],L)

4.  inserted(U,[V|L],[V|M]) :- inserted(U,L,M)
```

where 'inserted(U,L,L1)' has the intended interpretation that the list L is the list L1 with the element U

inserted at some (arbitrary) position.

Again these specifications could be claimed to have clear and acceptable declarative readings. We will assume that 'dsafe' can be equally nicely specified - it has no detailed pedagogic role to play in our example.

We seem then to have exploited the model theoretic semantics of Prolog to obtain a very clear and complete specification of a solution to the eight-queens problem by the set of Horn clauses 1.2.... Let us call this set of sentences "A" . The existence of a solution to the eight queens problem could now be asserted by the sequent "A => queens(Q)" .

Prolog would instantiate Q properly. However, if clause 1 of the specification were changed to the apparently equivalent clause

```
1.  queens(Q) :- perm(Q,[1,2,3,4,5,6,7,8]),dsafe(Q)
```

leaving everything else unchanged, then Prolog would <u>not</u> instantiate Q .

This remark is not intended as a criticism of Clark & McCabe, but to draw attention again to the difference between possible expectations and the reality. One has the expectation that a satisfactory axiomatization of 'perm' would necessary capture the symmetry of the relation. The Prolog implementation of the axiomatization by Clark & McCabe does not. In the context of its sole use in a particular axiomatization of the eight-queens problem, the effects of asymmetry have been nullified: in general, however, this might be treating the symptom rather than the disease and would become increasingly opaque in more complex problems involving deeper nestings of axiomatized relations.

The difficulty of the Clark & McCabe axiomatization lies in the fact that if one backtracks to a call of perm(M,L) where M is a variable, then clause 4 for inserted is repeatedly used, each use generating a candidate permutation M consisting of a list with one more uninstantiated variable at its head and an uninstantiated variable tail, each of which candidate permutations finally fails the call perm(M,[]) - as, of course, it should!

It might be thought this problem with 'perm' could be olved by using meta-logical features of Prolog in some pecification such as:

```
1.  perm(L,M) :- nonvar(L) , ! , perm1(L,M)

2.  perm(L,M) :- nonvar(M) , ! , perm1(M,L)

3.  perm1([],[])

4.  perm1(M,[U|L]) :- inserted(U,M,N) , perm1(N,L)

5.  etc., etc.
```

ith the intention that "perm1" is only called with an ppropriately instantiated argument pair such that "inserted" s well-behaved. This specification would certainly "solve" he original problem associated with the eight-queens pecification: however, the new specification of "perm" ehaves in a similar way to the first for pairs of calls such s "perm([1,2,3],[2|L])" and "perm([2|L],[1,2,3])".

In order to emphasize the point made earlier about the otivation of this brief note, a digression is in order. The ollowing is a quotation from comments by an unknown referee presumably chosen for his expertise) of an earlier version f this note:

> "...;for that matter any Prolog programmer knows or should know, that if he wants his predicate to work independently of the data flow he must be careful; hence program ... is not to be written if one knows that L can be a free variable or "infinite" (i.e. end up with a free variable); further the problem is not necessary with "perm", it can be argued that it is with "inserted ": one should then write:
>
> ```
> inserted(U,V,L) :-
>           not var(V), ! , insert(U,V,L)
> ```
>
> where "insert" is given by
>
> ```
> insert(U,[U|L],L)
>
> insert(U,[V|L],[V|M]) :- inserted(U,L,M)
> ```

Admittedly this is a partial solution, but is it correct in all cases where inserted is called; in particular perm([1,2,3],[2|L]) gives the two correct answers for L and M ; similarly for perm([2|L],[1,2,3]) which fails ... Hence there is a natural way to get it right." (My underlining - E.W.E.)

Adopting the referee's suggestion, the specification of "perm" becomes:

```
perm([],[])
perm(L,[U|M]) :- inserted(U,L,L1) , perm(L1,M)
inserted(U,V,L) :- not var(V), !, insert(U,V,L)
insert(U,[U|L],L)
insert(U,[V|L],[V|M) :- inserted(U,L,M)
```

Note that the referee has essentially addressed the subproblem of non-termination by making one of "perm(M,L)" and "perm(L,M)" fail! I did not and still do not regard this as a "natural" way to get "it" right!

Finally, to avoid potential misunderstandings let me stress that this somewhat curious digression has been made to emphasize that my men are not all straw! Some are flesh and blood and refereeing!

Returning to the symmetry problem: in all cases the incompleteness stems from the potential for generating objects from an infinite domain by backtracking. Both the specification of "mem" at the beginning of the paper and our specifications of "perm" give trouble for this reason.

In the case of "mem" the difficulty was removed by a reordering of clauses with the result that no new candidate was generated "unnecessarily". In the case of "perm" the problem is deeper: it cannot be solved by reordering nor by meta-logical wizardy - which indeed addressed the "wrong" problem. Equally important, this kind of incompleteness is potentially difficulty to detect - particularly when the calls to an offending m-ary relation are part of higher relations themselves possibly with completeness constraints of their argument tuples. Thus, in our introductory pedagogic context, the call of

"perm([1,2,3,4,5,6,7,8],Q)" comes from the body of the specification of "queens". Although, as already mentioned, once having diagnosed our difficulty, it is not onerous to change the call to "perm(Q,[1,2,3,4,5,6,7,8])" , one could easily construct more sophisticated examples where the choice of appropriate orderings of argument tuples could become quite a tricky problem.

In the case of our illlustrative example of "perm", and having identified that the difficulty stems from the potential for "inserted" in the specifications above to generate an infinite sequence of objects each of which possesses a property which is going to lead to failure, we can see that it is possible to respecify "perm" to make this impossible. There are two interestingly different ways to do this.

The property in the "perm" specification is that each of the generated lists, L , say, has a length one greater than its predecessor and that the initiation of backtracking takes place by a failure of "perm(M,[])"! Recognition of this motivates the specification:

1. perm(L,M) :- samelength(L,M) , perm1(L,M)

2. samelength([],[])

3. samelength([U|L],[V|M]) :- samelength(L,M)

4. < clauses specifying "perm1" as in Clark & McCabe specification above. say >

With this specification, any call of "perm" with an argument tuple which fixes the common (finite) length of the argument lists will lead to "inserted" being called in a context in which generation of an infinite sequence of objects cannot occur. For example, one of our earlier "problem" calls "perm([2|L],[1,2,3])" would now result in "perm1([2|L],[1.2,3])" being called in an environment in which L is bound to the list [X,Y]. In effect we have a call of "perm1([2,X,Y],[1,2,3])": which call does not permit infinitary generation.

One might, somewhat impudently, present the specification with the motivation that the intended procedural reading is to be "well, we'll do a quick check that the two argument lists are indeed globally consistent with the relation of permutation before we get down to crossing the i's and dotting the t's" !!! Indeed, in the case where L and M are both explicit finite lists then "samelength" acts like this (apart from the tongue in cheek adjective "quick"). More generally however "samelength" acts to construct the most general finite lists L and M which can satisfy "samelength(L,M)" and it has been introduced into the specification for just this reason.

A second (and more "honest"?) way to obtain an axiomatization which is symmetric under the Prolog interpreter is to use a subtler kind of redundancy and write the 'constructive' axioms:

```
perm([],[])

perm(X,Y) :- perm(X1,Y1) , inserted(U,X,X1) ,
             inserted(U,Y,Y1)
```

with the previous axiomatization of "inserted".

Here we simply have the embarrassment that each of the permutations is generated twice!

"This is a long cautionary tale" said the mouse.

The Prolog "perm" saga does not end here. What happens if one wants the set of permutations of some finite list say?

Sets of consequences in Prolog are handled by a non-logical operator "set-of" which essentially explores the whole potential sequent space aggregating appropriate instantions of variables in provable sequents.

Certainly "set-of" works with the axiomatization of "perm" using the covert "samelength" device within the domain of this device.

However, "set-of" does not work with the more "honest" axiomatization immediately above and for the same reason as for previous failures: "set-of" eventually enters an infinite search space, and we once again have the problem of non-termination.

We see here a subtle interaction in Prolog of incompleteness (in the obvious sense), and non-logical operators specifically introduced to do what otherwise couldn't be done!

An adjournment

Having got the bit between one's teeth and with the success of 'samelength' to motivate one, one can return to the original attempt at a direct symmetric specification of "perm" and try

```
perm([],[])
perm([X|L],[X|M]) :- perm(L,M)
perm([X|L],[Y|M]) :- inserted(X,M,M1,Y,L,L1) ,
          perm(L1,M1)
inserted(X,[X|M],M,Y,[Y|L],L)
inserted(X,[X|M],M,Y,[Y1|L1],[Y1|L2]) :-
          inserted(X1,M,M1,Y,L1,L2)
inserted(X,[X1|M1],[X1|M2],Y,]Y|L],L) :-
          inserted(X,M1,M2,Y1,L,L1)
inserted(X,[X1|M1],[X1|M2],Y,[Y1|L1],[Y1|L2]) :-
          inserted(X,M1,M2,Y,L1,L2)
```

where inserted(X,L,L1,Y,M,M1) has the intended interpretation that L(M) is the list L1(M1) with X(Y) inserted in it. This is Prolog - symmetric and works with "set-of". (Of course, one should prove these statements?)

The necessity for the 6-ary function and the multiplicity of cases in the model theoretic reading rather detract from any sense of achievement!

Summary

The late Christopher Strachey told the story that whenever he gave talks on his design for the language CPL, he would inevitably be asked "but can it do so-and-so?" Strachey claimed that as the designer of a good programming language there were

only two possible answers he could give and they were either "of course it can!", or "of course it can't!"

One of the difficulties with Prolog is that it does not meet this criterion and this note has attempted to give a simple example of an important way in which it fails. In a phrase: Prolog holds out promises it cannot fulfill. In particular, to have to consider potentially difficult proofs of termination of procedural readings of obviously true sequents, seems to run counter to one's intuition of what "logic programming" is all about. One of the reasons for the author's concern is that it is like Absys (Foster, 1969) in this. Examples of other ways in which Prolog fails to meet Strachey's criterion could be given. It may well be that there will emerge brands of logic programming languages that will be both pragmatically useful and which will indeed meet Strachey's criterion. It is consistent with the goals of much current research in Artificial Intelligence that the cause of difficulties arising from a first tentative specification might be automatically diagnosed and rectified by a suitable respecification and, as an issue in the study of knowledge representation and use, some form of the problems identified with Prolog above will have to be faced as part of that study as such.

In the absence, however, of substantive progress on such issues it might be better to recognize that the goals of logic and the goals of programming should be regarded as essentially different unless proven otherwise. The goal of logic as usually conceived is to exhibit what things follow from what. The goal of programming as usually conceived is to exhibit how to construct something from other things. Although "what follows from what" certainly provides the framework in which a construction is demonstrated to be valid, to call this validation process "control", at least in the simplistic sense of current computational control structures, and to regard Prolog programming as "logic plus control" e.g. Kowalski, 1979, is to stretch a good catchphrase too far. In what sense, for example, is it appropriate to regard the 'samelength' assertion as a control component of the specification of 'perm' in the example above? Like

most provoking catchphrases, "programming as logic plus control" can be given interesting interpretations. However, for now a better catchphrase, if one wants catchphrases at all, might be that "logic is (Prolog) programming minus control" : a Prolog program, [A,G] , terminating or not, stripped of a particular procedural semantics with its particular concomittant 'control', and re-interpreted as a sequent A => G , is always, if true, demonstrably true in first order logic. (The asymmetry in the two catchphrases is, of course, only in the eye of the believer!).

As mentioned in the introduction, although the declarative aspect of computational text is very important and has been increasing illuminated by the study of the relation between logic and programming, the relationship between consequence and construction is very subtle, and its subtlety must be respected.

The content of this note and, in particular, the relationship between consequence and construction, is being elaborated in a further technical report in preparation. The work is being conducted under Operating Grant Number A9123 from the Natural Sciences and Engineering Research Council of Canada.

References

1. Clark, K.L. & McCabe, F.G. (1979). The control facilities of I.C. Prolog, Expert Systems in the Micro Electronic Age (D. Michie, ed), Edinburgh University Press.

2. Clocksin, W.F. and Mellish, C.S. (1981). Programming in Prolog. Springer-Verlag. Berlin.

3. Foster, J.M. and Elcock, E.W. (1969). Absys 1: an incremental compiler for assertions: an introduction. Machine Intelligence 4, (Eds. Meltzer, B. and Michie, D.). Edinburgh University Press. Edinburgh. pp. 423-429.

4. Kowalski, R.A. (1979). Algorithm = Logic + Control. C.A.C.M. Vol. 22, No. 7. pp. 424-436.

A polymorphic type system for Prolog

Alan Mycroft
Dept of Computer Science
Edinburgh University

Richard O'Keefe
Dept of Artificial Intelligence
Edinburgh University

Abstract

We describe a polymorphic type scheme for Prolog which makes static type checking possible. Polymorphism gives a good degree of flexibility to the type system, and makes it intrude very little on a user's programming style. The only additions to the language are type declarations, which an interpreter can ignore if it so desires, with the guarantee that a well-typed program will behave identically with or without type checking. Our implementation is discussed and we observe that the type resolution problem for a Prolog program is another Prolog (meta-) program.

1 Introduction

Prolog currently lacks any form of type checking, being designed as a language with a single type (the term). While this is useful for learning it initially and for fast construction of sketch programs, it has several deficiencies for its use as a serious tool for building large systems.

We have observed that a theorem prover which reasons about Prolog programs can be more powerful if it has type information available. One indication as to why this is so can be seen from the fact that the traditional definition of *append* has append(nil,3,3) deducible from its definition.

One very good reason for a type system is that it can provide a static tool for determining whether all the cases in a Prolog predicate have been considered. For example, a predicate defined by

```
type neglist(list(int),list(int))
neglist(cons(A,L),cons(B,M)) ← negate(A,B), neglist(L,M)
```

will never succeed, since we have probably omitted the clause

```
neglist(nil,nil) ←
```

A type system would enable us to detect this by checking for exhaustive specificatior of argument patterns for a given data-type. Of course, if we really did want a certain case to fail, then adding a clause such as

```
neglist(nil,nil) ← fail
```

would be an explicit way of requesting such an event without leaving first-order logic (and would facilitate later reading of the program).

Moreover, our type system can be used as the basis of an encapsulatior providing an abstract data type facility. The ability to hide the internal details of a given object greatly aids the reliability of a large system built from a library o modules.

Finally, we note that static type checking cannot of itself provide a great increase in speed of Prolog programs, due to the fact that term unification must still be performed, as in the dynamic case. However, typed Prolog can improve the speed of compiled clauses of a given predicate by using a mapping of data constructors onto small adjacent integers to enable faster selection of the clause(s) to be invoked. By far the greatest gain is that of programmer time provided by early detection of errors.

As far as we know this work is the first application of a polymorphic type scheme to Prolog, but related work includes Milner's work [4] on typing a simple applicative language which is used in the ML [3] type checker and the HOPE language which uses a version of Milner's algorithm extended to permit overloading. However, this work differs from these in several respects. Firstly, the formulation of Prolog as clauses means that the problems of generic and non-generic variables are much reduced. All predicate and functor definitions naturally receive generic polymorphic types which can be used at different type instances within the program whilst all variables receive non-generic types. Moreover, our formulation for Prolog removes a restriction in Milner's scheme in which all mutually recursive definitions can only be used non-generically within their bodies. Thus in ML the (rather contrived) program

```
let rec l x = x
    and f x = l(x+1)
    and g x = if l(x) then 1 else 2
```

would be ill-typed. Since all Prolog clauses are defined mutually recursively, this restriction would have the effect of making the polymorphism useless.

2 Mathematics

We assume the notion of substitution, a map from variables (and terms by extension) to terms, ranged over by θ and ϕ. An invertible substitution is called a renaming. If a term, u, is obtained from another, v, by substitution then we say that u is an instance of v, and write $u \leqslant v$. We write $u \cong v$ if $u \leqslant v$ and $v \leqslant u$. This means that u and v only differ in the names of their variables and that the substitutions involved are renamings. Also assumed is the notion of most-general unifier (MGU) of two terms.

For any class of objects S, the notation S^* will be used to indicate the class of objects consisting of finite sequences of elements of S.

3 Prolog

The simple variant of Prolog we consider will be defined by the following syntax (we assume the existence of disjoint sets of symbols called Var, Pred and Functor, representing variables, predicates and functors symbols respectively):

```
Term     ::= Var | Functor(Term*)
Atom     ::= Pred(Term*)
Clause   ::= Atom ← Atom*
Sentence ::= Clause*
Program  ::= Sentence: Atom
Resolvent ::= Atom*
```

By definition of clause form each, implicitly universally quantified, variable appears in at most one clause. To make the formal description of typing simpler, we assume that the textual names of variables also follow this rule. A program then is given by a finite list of (Horn) clause declarations, followed by an Atom (short for atomic formula), called the query, to evaluate in their context. It specifies an initial resolvent by taking the query and treating it as a one-element list.

The evaluation mechanism for Prolog is very simple, and based on the notion of SLD-resolution as the computation step:

SLD-resolution is the one-step evaluation which transforms a Resolvent. Given a resolvent

$$R = A_1, \ldots, A_n$$

we select an Atom, the selected atom, say A_k, (this is often A_1 in real Prolog interpreters) and perform resolution with it and a matching clause. So, choose a clause of the program, the selected clause, say Q, given by

$$C \leftarrow B_1, \ldots, B_m$$

and suppose that R has no variables in common with it (otherwise we must rename its (Q's) free variables since they are implicitly universally quantified for the clause).

Now let θ be $MGU(A_k, C)$ if this exists. If it does, then we can rewrite R into R' given by

$$\theta(A_1, \ldots, A_{k-1}, B_1, \ldots B_m, A_{k+1}, \ldots, A_n).$$

The most common form of Prolog interpreter uses k=1 when this expression simplifies somewhat.

An answer is produced when the resolvent is rewritten into a sequence of zero atoms. The associated answer to such a rewriting sequence is the composition of most-general unifiers encountered during the rewriting process, or rather its restriction to the variables in the query.

Observe that the above specification only told us how we could produce an answer (if one exists) from a Prolog program. For computation the choices above (the selected atom and clause) must be incorporated into a deterministic tree searching algorithm, which we take time to explain below for the reader's benefit. However, we would like to stress now that the results on type-checking given in section 5 work for any order of evaluation (choices of atoms and clauses) of Prolog programs (depth-first/breadth-first/coroutining/parallel).

3.1 Digression: SLD-trees

The idea of SLD-resolution above, leads to the idea of an SLD-tree: whenever we are forced to select a clause then, instead of irreversibly choosing a given matching clause, we construct a tree of resolvents (an SLD-tree) where a resolvent has a son resolvent for each clause which matches with the selected atom. A sensible computation (the standard implementation of Prolog) is then to search this tree in depth-first left-right manner.

Some branches die out, in that no clause matches the selected atom, whereas others have more than one subtree contribute to the answer. This is often referred to as the non-determinacy of Prolog.

Finally, we remark that there is never any need to seek alternatives to the selected atom - in fact doing so would merely lead to duplication of answers exhibited elsewhere in the SLD-tree. (For more details on this aspect see [1]).

4 Types

The scheme of types (Type) we allow are given by the following grammar and are essentially the same as those which occur in ML [3]. We assume disjoint sets of type constructors (Tcons, ranged over by roman words) and type variables (Tvar, ranged over by greek letters like α, β, γ). These are also assumed to be disjoint from Var, Pred and Functor.

Type ::= Tvar | Tcons(Type*)

Type will be ranged over by $\rho\sigma\tau\ldots$. A type is called a monotype if it has no type variables. Otherwise it is a polytype.

For examples, we suppose that Tcons includes the nullary constructor int and the unary list. Example types are then

list(α), int, list(list(int)), etc.

Note that the third type is an instance of the first.

4.1 Digression: the Unary Predicate Calculus

The type systems used in many AI programs are variants or restrictions of the Unary Predicate Calculus. However, UPC is not adequate as the single type system for an AI programming language. Rules such as

$$(\forall\ N, L)\ \text{integer}(N)\ \&\ \text{int\_list}(L) \Longrightarrow \text{int\_list}(\text{cons}(N, L))$$
$$(\forall\ L)\ \text{int\_list}(L) \Longrightarrow (L=\text{nil}\ \vee\ \text{integer}(\text{car}(L)))$$

cannot be expressed in it.

5 Well-typing of Prolog

This section contains the central definition of a Prolog program being well-typed together with precursor and auxiliary definitions. Many of the ideas appear in [4 where a polymorphic applicative language is typed, but our formulation for Prolo poses new problems and simplifies old ones as we discussed in the introduction.

Let Q the clause $C \leftarrow B_1, \ldots B_m$ and P be a finite subset of VarUPredUFuncto containing all the symbols of Q. We define a typing $\bar{P}$ of P to be an association o an extended type to each symbol occurring in Q. The types are members of a give algebra as defined in section 4. Predicates and Functors are associated wit extended types as given below. Types and extended types will be written as superscript on the object they are associated with. σ_i and τ will represent (non extended) types. For each variable X occurring in Q, $\bar{P}$ will contain an element o the form X^{τ}. For each predicate a of arity k in Q, $\bar{P}$ will contain an element of th form $a^{\sigma_1,\ldots,\sigma_k}$. For each functor f of arity k in Q, $\bar{P}$ will contain an element of th form $f^{(\sigma_1,\ldots,\sigma_k)\to\tau}$.

Similarly, the clause Q will be written as a typed clause $\bar{Q}$ by the writing of a typ on each term (this includes variables).

As an example of a clause and its typing consider the clause Q, given by

$$\text{app}(\text{cons}(A, L), M, \text{cons}(A, N)) \leftarrow \text{app}(L, M, N)$$

The set P = {A, L, M, N, app, cons} gives its set of symbols, and a typing (which wi turn out to be a well-typing considered later) can be given by $\bar{P}$:

$$(A^{\alpha}, L^{\tau}, M^{\tau}, N^{\tau}, \text{app}^{\tau,\tau,\tau}, \text{cons}^{(\alpha,\tau)\to\tau})$$

where τ is used for a shorthand for list(α) and the associated clause typing $\bar{Q}$ given by:

$$\mathrm{add}(\mathrm{cons}(A^{\alpha}, L^{\tau})^{\tau}, M^{\tau}, \mathrm{cons}(A^{\alpha}, N^{\tau})^{\tau}) \leftarrow \mathrm{add}(L^{\tau}, M^{\tau}, N^{\tau})$$

$\bar{P}$ will be called the typed premise of $\bar{Q}$ due to the relation to theorem proving.

Fortunately, it will turn out that most of the mess of types written above are inter-dependent and the above expression can be well-typed much more succinctly - see later.

We will now define $\bar{Q}$ to be a well-typing of Q under $\bar{P}$, written $\bar{P} \vdash \bar{Q}$ if the following conditions hold:

1. $\bar{P} \vdash (A \leftarrow B_1, \ldots, B_m)$ if
   $A = a(t_1^{\tau_1}, \ldots, t_k^{\tau_k})$ and $a^{\rho} \in \bar{P}$
   with $(\tau_1, \ldots, \tau_k) \cong \rho$
   and $\bar{P} \vdash t_i^{\tau_i}$ $(1 \leq i \leq k)$
   and $\bar{P} \vdash B_i$ $(1 \leq i \leq m)$.

2. $\bar{P} \vdash A$ if A is an Atom and
   $A = a(t_1^{\tau_1}, \ldots, t_k^{\tau_k})$ and $a^{\rho} \in \bar{P}$
   with $(\tau_1, \ldots, \tau_k) \leq \rho$
   and $\bar{P} \vdash t_i^{\tau_i}$ $(1 \leq i \leq k)$.

3. $\bar{P} \vdash u^{\sigma}$ if u is a Term and
   $u = f(t_1^{\tau_1}, \ldots, t_k^{\tau_k})$ and $f^{\rho} \in \bar{P}$
   with $((\tau_1, \ldots, \tau_k) \rightarrow \sigma) \leq \rho$
   and $\bar{P} \vdash t_i^{\tau_i}$ $(1 \leq i \leq k)$.

4. $\bar{P} \vdash X^{\sigma}$ if $X^{\sigma} \in \bar{P}$.

Now, we will define a program to be well-typed under a typed premise $\bar{P}$ if each

of its clauses is well-typed under $\bar{P}$ and if its query atom is. Similarly a resolvent is well-typed if each of its atoms are.

Well-typing as a mathematical concept is of little use, unless we relate it to computation. This we will now do, under the motto "Well-typed programs do not go wrong".

6 Well-typed programs do not go wrong

What we desire to show, is the semantic soundness condition that if a program can be well-typed, then one step of SLD-resolution will take a well-typed resolvent into a new well-typed resolvent. Thus any SLD-evaluation of a well-typed program will remain well-typed. It is trivially the case that the initial resolvent is well-typed if the program is. Moreover, we should show that the variables in the query can only be instantiated to terms specified by their types given by the well-typing.

The first condition is simply proved: Let R be the resolvent $A_1, \ldots, A_n$ and let Q be a clause $C \leftarrow B_1, \ldots, B_m$ which has no variables in common with R (the case where Q and R have variables in common will be discussed later). Without loss of generality (symmetry) let A_1 be the selected atom and suppose $\theta = MGU(A_1, C)$ exists. The resolvent produced by one-step evaluation is R' given by

$$\theta(B_1, \ldots, B_m, A_2, \ldots, A_n).$$

We will now show how to well-type this from the well-typing of R.

Let us suppose that there is a P with typing $\bar{P}$ and associated well-typings $\bar{R}$ and $\bar{Q}$ such that $\bar{P} \vdash \bar{R}$ and $\bar{P} \vdash \bar{Q}$ (note this provides well-typings $\bar{A}_i$, $\bar{C}$, $\bar{B}_i$). Moreover let us suppose that $\bar{R}$ and $\bar{Q}$ have no type variables in common (again, we will discuss this later, but note that the typing rules never rely on the 'absolute' names of the type variables).

Let the type of the predicate symbol of C in $\bar{P}$ be $c^{\rho_1, \ldots, \rho_k}$. Now the well-typing determines that $\bar{C}$ can be written $c(s_1^{\sigma_1}, \ldots, s_k^{\sigma_k})$ and $\bar{A}_1$ as $c(t_1^{\tau_1}, \ldots, t_k^{\tau_k})$ where

$$(\sigma_1, \ldots, \sigma_k) \cong (\rho_1, \ldots, \rho_k)$$
$$(\tau_1, \ldots, \tau_k) \leq (\rho_1, \ldots, \rho_k).$$

This means that there is a substitution ϕ on type variables (actually $\phi \cong MGU((\sigma_1, \ldots, \sigma_k), (\tau_1, \ldots, \tau_k)))$ such that $(\tau_1, \ldots, \tau_k) = \phi((\sigma_1, \ldots, \sigma_k))$.

The claim is that

$$\bar{P} \vdash \theta(\phi(\bar{B}_1), \ldots, \phi(\bar{B}_m), \bar{A}_2, \ldots, \bar{A}_n)$$

gives a well-typing of R', where applying ϕ (a type substitution) to a typed atom means that it is to be applied to the type variables in types associated with terms occurring within that atom.

We now address the problem of there being variables, or type variables, in common between R and Q. These are really the same problem (the perennial one of renaming in Prolog). A simple solution is the following: Whenever we come to perform resolution between a clause Q and a resolvent R we rename Q such that all its variables (using a renaming ψ) and all its type variables (using a renaming η) are distinct from the variables (and type variables) in R and the other clauses. This can always be done since R can only contain a finite number of different variables. Moreover this does not change the meaning of Q. This strictly breaks the type scheme, since the new variables appearing in Q do not appear in $\bar{P}$. However, a simple addition to $\bar{P}$ of $\psi(X)^{\eta(\tau)}$ for each variable X in the original Q which appeared as X^{τ} in $\bar{P}$ serves to correct this and preserve the typing. We are now back in the case where Q and R have no variables or type variables in common.

We now return to the problem of showing that a well-typed program can only instantiate the variables of its query to values having types as dictated by the typed premise. To see that this is the case, it is merely necessary to observe that each resolution step (as above) is performed between an Atom, A, and a (type) instance of a clause $C \leftarrow B_1, \ldots, B_m$. such that the types of A and this instance of C are identical except for the names of type variables. Thus variables in A can only be instantiated to Terms (possibly other variables) having identical types. The whole result is proved by induction on the length of computation leading to a refutation.

7 Specification of the type information to Prolog

We suggest that the type specification be performed by annotations to the Prolog system. The well-typing required three sets of information to be supplied:

- the types of the predicates
- the types of the functors
- the types of the variables.

We suggest that declarations be supplied which give the type of the first two but the type of variables can easily be determined from them. This can be seen by observing that a well-typed Atom or Term labels the type of each argument Term, and so each variable is labelled with a type. The most-general unifier of all the types associated with a single variable (if it exists) gives a type for that variable. (This is also convenient since the scope of variables in Prolog is a single clause, whereas the other objects have a global scope.)

It is convenient to specify the names of types along with the functors which create them from other types. This has been demonstrated by HOPE [2] and we do not expect to better this idea.

So one of the declarations, or meta-commands is one of the form

Declaration ::= 'type' Tcons(Tvar*) '=>' Functor(Type*)*.

Examples would be (the second somewhat improper)

type list(α) => nil, cons(α, list(α))
type int => 0, 1, -1, 2, -2, 3, -3, ...

The second declaration specifies the type of predicates. Suggested syntax is

Declaration ::= 'pred' Pred(Type*).

and an example for the 'equal' function defined by

equal(X,X) ←

would be

pred equal(α, α).

We note here, that, given the types of the functors, then would seem possible to determine the types of the predicates involved without any great amount of work (as in ML [4]). However, this seems to depend on an analysis of the whole program at once, rather than any form of interaction.[1] We would also claim that the documentation provided by the written form of the types facilitates human understanding of programs in much the same way that explicit specification of mode information (input/output use of parameters) for predicates does.

7.1 Abstract data types

We observe that the above declarations furnish a form of abstract data typing. Providing a 'module' construct and exporting from it a given type name, and predicates which operate on that type, but not the constructor functors for that type, enables us to use a type, but not to determine anything about its representation. HOPE has such a construct, and we think it would greatly benefit Prolog.

8 Overloading

The above discussion has centred on a formalism for well-typing Prolog. However, it does not allow for one feature which we have found to be useful, and which is very easy to build into the type system. This feature is overloading and appears in a similar form in HOPE [2].

The observation, is, that quite often, we may wish a given function, predicate or functor name to stand for more than one distinct operation. This is common in mathematics and computer science, where an operator (eg '+') may be used to denote a different function at different types. In Prolog this can be useful too. For example, we may wish to have types specified by

list(α) => nil, cons(α, list(α))
tree(α) => nil, leaf(α), cons(tree(α), tree(α))

where the constructors nil and cons(_,_) have different meanings according to whether they act on lists or trees. (Of course we could give them different names,

---

[1] Moreover there is a small technical problem concerning recursive definitions which makes checking of type specifications of such definitions much easier than their derivation.

but this is not always helpful to the programmer.)

Similarly, we may want certain predicate symbols to refer to different predicate according to the type of their arguments. A typical example would be some sort c 'size' predicate.

We formalise this by permitting the typed premises used above to contain mor than one type associated with any given functor or predicate symbol.

9 Implementation

We have built such a system in Prolog which implements the overloaded typ checker by backtracking. Note that this is not particularly difficult since our well typing rules given in section 5 are essentially Horn clauses. There are merely tw points to observe. Firstly, the 'occur-check' of unification (which is often omitted b Prolog implementations) is essential for this typechecking scheme. Secondly, th use of $\leqslant$ can be simulated by instantiation of a copy of the functor or predicate typ and the use of $\cong$ by a common meta-linguistic predicate (numbervars) whic instantiates variables in a term to ground terms to avoid their further instantiation Copies of the code can be obtained from the authors or could be included as a appendix.

That the well-typing rules (which define when a given program has a given type can be used to determine the type of a given program is a simple consequence c the Horn clause input/output duality. Moreover, when the well-typing rules are use in the fashion on a given program, T say, then the standard SLD-resolution wi produce a terminating evaluation giving the most-general types associated with T The basic idea is that if the well-typing problem has no solution, ~~the~~ the program i ill-typed. If it has exactly one ~~the~~ the program is well-typed, and if it has more tha one then some overloaded operator is ambiguous.

10 Higher order objects

This section is much more tentative and more in the manner of suggestion tha the rest of the paper and we would be grateful for any comments on its inclusion c its contents. It is included because we want to discuss the well-typing of objects whic

do not form part of first-order Prolog, in particular the call and univ operators.

The definition of call is based on the fact that most Prolog implementations use the same set of symbols for predicates and functors (this causes no syntactic ambiguity) and thus a Term has a naturally corresponding Atom. Hence call is defined to be that predicate such that call(X) is equivalent to Y where Y is the Atom corresponding to the Term X. Thus call provides a method of evaluating a Term which has been constructed in a program and is accordingly related to EVAL in LISP. We would like to argue that such a predicate is more powerful than is required and indeed encourages both bad programming style and inefficient code. It is certainly the case that most uses of call are used in the restricted case of applying a certain functor passed as a parameter to arguments determined locally (as in mapping predicates). Functions or predicates like EVAL or call do not appear to have sensible types and are thus generally omitted from strongly typed languages in favour of some form of APPLY construct.

We would like then to change our definition of Prolog and its typing to introduce this construct. To do this we introduce a family of abstract data types, called

pred(α), pred(α,β), pred(α,β,χ), ...

and a family of predicates with types given by

pred apply(pred(α),α), apply(pred(α,β),α,β), ...

The only way to introduce object of type pred is by a special piece of syntax given by

Term ::= `Pred

which has the effect of associating the definition of the given predicate with Term, which then receives the type pred(α_1,...,α_n) if the predicate has type (α_1,...,α_n). (It may be desirable to use such syntax as `foo/3 or `foo(_,_,_) if several predicates of different arities have the name foo.) Such values can only be used in apply and have the effect of using the associated predicate value together with Terms as arguments to produce an Atom to evaluate. It can be shown that such a scheme is type secure. For example, the map predicate can be defined and used by:

120

```
map(F,cons(A,L),cons(B,M)) ← apply(F,A,B), map(F,L,M)
map(F,nil,nil) ←
neglist(X,Y) ← map(`negate,X,Y)
```

assuming that negate is defined as a diadic predicate. The type of map so defined would be (pred(α,β),list(α),list(β)).

The other higher-order object frequently used is the univ predicate (often written '=..') which can be used to transform a Term into a list of Terms derived from the former's top-level substructure. (This is typically used for analysing terms read with input functions.) Thus

```
univ(f(g(X,a),Y), [f,g(X,a),Y])
```

is true. As it stands this clearly breaks the type-scheme we are proposing since the elements of the list represented by the second parameter need not be of the same type. We observe again, that such a predicate is not commonly used in its full generality, but rather to allow arbitrary terms to be input. As such, we suspect that introducing a new type 'input_term' which specifies the type of objects generated by input routines and giving univ the type (input_term,list(input_term)), together with a notation for treating a Term as an input_term would give much of the power of univ within a strong typing discipline.

The difficulty of typing univ arise from the conflation of object and meta levels in one language, which requires the same object to simulatenously possess at least two types, in a stronger sense than overloading. A satisfactory resolution of this problem waits on the introduction of an explicit meta-level or the construction of a genuinely reflective Prolog [5].

11 Conclusions and further work

We have shown how to well-type that subset of Prolog described by first-order logic and indicated how this might be extended to allow higher order objects. It is an interesting result that the well-typing problem for a Prolog program can itself be regarded as a Prolog meta-program.

121

12 Acknowledgments

This work was supported by the British Science and Engineering Research Council.

13 References

[1] Apt, K.R. and van Emden, M.H.
Contributions to the theory of logic programming.
Journal of the ACM 29(3):841-862, 1982.

[2] Burstall, R.M., MacQueen, D. and Sannella, D.T.
HOPE: an Experimental Applicative Language.
In Conference Record of the 1980 LISP Conference. 1980.
Also internal report CSR-62-80, Dept. of Computer Science, Edinburgh University.

[3] Gordon, M.J.C., Milner, A.J.R.G., Morris, L., Newey, M. and Wadsworth, C.
A Metalanguage for Interactive Proof in LCF.
In Proc. 5th ACM Symp. on Principles of Programming Languages. Tucson, Arizona, 1978.

[4] Milner, R.
A Theory of Type Polymorphism in Programming.
Journal of Computer and System Sciences 17(3):348-375, December, 1978.

[5] Smith, B.C.
Reflection and Semantics in a Procedural Language.
PhD thesis, MIT LCS, 1982.

122

Table of Contents

PRISM
A Parallel Inference System for Problem Solving

Simon Kasif Madhur Kohli Jack Minker

Department of Computer Science
University of Maryland
College Park, MD 20742
(301) 454-4251
MINKER @ UMCP-CS

Abstract

A Parallel Inference System for Problem Solving (PRISM) has been developed at the University of Maryland. The system is designed to provide a general experimental tool for the construction of large artificial intelligence problem solvers.

We present some of the basic facilities for controlling parallelism and inference provided by the system. PRISM is based on the concept of logic programming with a separate control componenent. The control may either be explicitly specified by the user in his input or alternatively determined dynamically by the system, which takes advantage of the implicit parallelism in the logic of the algorithm. The design makes the underlying virtual architecture transparent to the user. The system supports both AND and OR parallelism.

1. Introduction and Overview

1.1. Introduction to Parallel Problem Solving

In general, problem solving systems have been designed to be executed on sequential machines (i.e. a single processor architecture). However, the complexity of many interesting problems, makes the sequential implementation of these problems infeasible in terms of speed and resource requirements. This implies that it is necessary to examine solutions to these problems in a distributed environment, in order to determine if these solutions will prove more feasible in terms of speed and resources, than those in a sequential environment. Furthermore, the investigation of distributed methods of problem solving is suggested by the structure of the problems themselves. Many interesting AI problems are NP-complete and require exponential time on a deterministic machine, whereas they can be solved in polynomial and sometimes linear time on a nondeterministic machine. A distributed system is necessary, to implement, nondeterministic solutions.

A large amount of work has been done on parallel architectures [Computer 1982a], [Computer 1982b] and algorithms for parallel architectures [Kung 1980]. Work has also been done on parallel languages and environments for parallel architectures for non-AI problems [Hewitt 1977], [Kahn 1977].

In the AI environment several researchers have suggested methods to parallelize certain types of problems, however, few of these schemes have actually been implemented on a distributed system. Kornfeld [1979], and Lieberman [1981] describe systems and languages which have been designed for parallel applications.

PRISM (a Parallel Inference System), which is an experimental tool for the development of distributed AI problem solvers, has been developed at the University of Maryland and has been implemented on ZMOB (Rieger [1980]). PRISM is based on logic programming (Kowalski [1979]).

1.2. Control in Logic Programs

In conventional programming systems the logic and control of an algorithm are combined making it difficult to separate or to modify control without affecting the logic. Logic as the specification language, is neutral with respect to control and specifies only the problem semantics. The method of how the problem is to be solved is external to the logic specification. It has been shown (Kowalski[1979], Pereira[1978], van Emden[1976]) that the complete separation of logic (the specification to be executed) and control (the order in which tasks are executed) allows a great amount of flexibility during execution, thus providing a natural parallel implementation of a program.

This is true since the inherent nondeterminism of logic programs can be exploited in many different directions during excution.

1. Top-down and bottom up execution of a program can be done in parallel.
2. At any time during execution more than one possible goal node (procedure) can be invoked.

3. Since the order of execution of atoms in a goal is usually not specified we can sometimes separate the goal into several independent subgoals to be solved.

4. Logic programs are distinguished from other applicative languages such as LISP due to the fact that more than one procedure can match a procedure call. This seeming disadvantage on a sequential machine becomes an advantage in a highly parallel environment since all or some matching procedures can be executed in parallel.

Thus, a primary issue in achieving a parallel system is developing an effective control specification that exploits parallelism. PRISM permits us to specify the problem independently of the control and allows us to experiment with alternative control possibilities for the same problem.

1.3. ZMOB and Parallel Problem Solving

PRISM has been implemented on ZMOB, which consists of a set of 256 Z80A microprocessors connected on a conveyor belt together with a host VAX-11/780 minicomputer. A description of ZMOB is given in the following section. A description of how parallel problem solving is achieved using ZMOB is described in Section 1.3.2.

1.3.1. ZMOB Description

The particular system to be used is ZMOB, a parallel multi-microprocessor system developed at the University of Maryland (Rieger[1980]). ZMOB is to consist of 256 Z80A microprocessors connected to a host computer (VAX 11/780) which is to communicate between machines via a high speed 48 bit wide, 257 stage shift register called the "Conveyor Belt" (Figure 1). The system is described in detail in Rieger[1980, 1981a, 1981b]. We shall briefly describe here only the communication features necessary to support PRISM.

The Z80A is a microprocessor capable of executing 400,000 instructions/second and has a 64K byte memory. Thus, the whole system is theoretically capable of executing 100 million instructions/second and has a memory capacity of 16 million bytes. Each processor is connected to the conveyor belt via a collection of high-speed 8-bit I/O registers and associated control circuitry, called the "Mail Stop". The registers are in charge of interrupt control, buffering and address control functions.

In general, the Conveyor Belt moves 257 bit patterns (bins) each 48 bits wide. Each processor can theoretically consume any bin that is currently at its mail stop, but it can send out information only in its own bin. The 48-bit message in the bin consists of four fields:

CONTROL	DATA	SOURCE	DESTINATION
8 bits	16 bits	12 bits	12 bits

Figure 1

The control bits allow the implementation of several communication strategies : Let (C X S D) be the content of a bin on the Conveyor Belt, then different control bits specify the following communication formats.

1. Direct addressing - The message X is sent to a processor whose physical address is D.
2. Pattern matching - Message X is sent to the first processor whose pattern (determined by Capability Code and Mask Registers in the Mail stop) matches D.
3. Send to all Processors - Message X is sent to all processors.
4. Send to a set of Processors - Message X is sent to all processors whose patterns match D.

Additionally, different settings of Control Registers in the Mail Stop allow the following :

5. Exclusive Source - This mode provides exclusive conversation between two processors and disables interrupts from other processors.
6. Readback - This mode allows an individual processor to intercept any of its own messages that went around the conveyor belt and was not consumed by any of the destination processors.

The following examples illustrate the utility of the above formats.

(3,5) Permits large blocks of data to be sent in a burst mode to all processors from the host computer. (e.g. to load kernel programs or data to all processors).

(2) Provides the ability to assign to each processor a relation. Logically the relation's name would be the pattern identifying this processor.

(4) Allows a very useful provision of clustering the system into independent sets of logically equivalent processors.

(6,4) Can be used to send a message to a set of processors and in case it was not consumed to activate a recovery routine.

1.3.2. ZMOB Parallel Problem Solving System

We find it useful to separate the static set of clauses representing the logic of a problem from the control which generates a search tree by applying these clauses to a goal clause. This distinction will be seen to be useful in experimenting with the control of a parallel logic programming system.

In particular we shall distinguish three separate portions of the system to which we dedicate microprocessors. These are :

(1) the problem solver (PS),

(2) the extensional database (EDB), the set of assertions, and

(3) the intensional database (IDB), the set of procedure clauses.

The PS administers the search space which consists of a tree of goal clauses. The root of the tree is the original goal, whereas successors of any clause C in the tree are resolvents obtained by resolving program clauses (EDB or IDB clauses) with an atom selected in C. Each leaf node in the tree is either the empty clause; or some indication that the respective branch of the search resulted in a failure; or an open goal clause not yet selected for expansion; or an active clause sent for expansion and not fully expanded.

To generate the successors of an open clause C the PS has to select an atom and send it to that part of the system that handles unification of the atom with procedure heads in the EDB or the IDB. If the tree is distributed among several microprocessors, several atoms of different clauses can be selected simultaneously for expansion. Atoms sent to the EDB/IDB for solving cause the return of information necessary for generating all successor clauses of C.

While waiting for the information the PS in each machine can treat other open clauses in the same way, so that the subproblems of several open nodes in the same machine can be solved in parallel and independent of each other.

A second part of the system is in charge of the assertions and procedure clauses. This is subdivided further into the extensional database (EDB) consisting of all function-free ground assertions, and the intensional database (IDB) that constitute the procedure clauses and non-EDB assertions (i.e. those that contain variables and/or functions).

This distinction was drawn primarily for two reasons. First, the EDB and IDB can use different unification algorithms. In particular, when matching an atom against an EDB entry, it is not necessary to invoke the occur check which is used to determine if a term substituted for a variable contains the variable. Second, there are many applications where the sizes of the EDB and IDB differ considerably. If the set of clauses is used as a database, the number of IDB clauses is likely to be relatively small, whereas there are many EDB clauses corresponding to a relational database in the usual sense. If, on the other hand, the set of clauses represents a program, there are usually few EDB-clauses, but the IDB clauses are generally numerous. In some instances we may wish not to make a distinction between EDB/IDB clauses. We want the system to be sufficiently flexible to be able to react in different ways.

In addition to predicates contained in the EDB and IDB, systems usually contain predefined predicates, e.g. arithmetic predicates or equality predicates. Such atoms are evaluated directly in the PS where encountered and are not sent to the EDB or IDB for evaluation.

Problems can arise if predicates are permitted to have side effects. One such side effect would be the ability to modify the database as, for example, contained in the PROLOG primitives ASSERT and RETRACT. As the system pursues different branches of the search tree in parallel, there is no way of determining the exact point at which the side effects were executed. Since side effects in one branch can influence other branches of the search tree, this fact would render the overall behaviour of the system intolerably unpredictable.

For that reason in this first design, we do not allow any predicates with side effects in a goal clause (and hence they are not permitted in a procedure clause) thus restricting the system to pure logic. This means that such

extralogical tricks as modifications to the database to simulate global variables are not permitted. Of course, the system must provide features other than the ability to solve goal clauses, including such capabilities as adding, deleting, and modifying clauses. Such capabilities are provided at the top level only, so that there is no modification of a knowledge base during problem solving.

The separation of the problem solving system into the problem solver, EDB search, IDB search, IDB monitor and VAX has isolated the functions in the system and has placed them on separate processors. The main link between the processors is the conveyor belt and message passing. There is an uniform message passing facility between machines.

2. Control Issues

2.1. Problem Solving Process

The problem solving process may be outlined as follows:

(1) the problem to be solved is expressed as a conjunction of goals, each of which is a subproblem to be solved;

(2) one or more subgoals may be selected to be solved;

(3) a subgoal is solved if it is matched by some assertion, or it is matched by a procedure which consists of a set of subgoals which can be solved.

The repeated execution of steps (1), (2) and (3) results in a top-down execution of a problem. One can specify a problem solving process which permits bottom-up, middle-out, top-down, or any combination of these reasoning methods. The initial PRISM system is restricted to top-down reasoning (backward chaining from the goal).

2.2. Goal tree and Control Issues

A goal tree is generated in the problem solving process. The goal tree is formed initially by placing the conjunction of goals to be solved in the root node of the tree. In general the tree consists of a set of nodes, where each node consists of a set of goals. Now, given a node, there are several ways in which the node may be executed. One or more goals may be selected to be executed asynchronously. This possibility provides for user control of parallel execution. Subgoals in a node may be characterized to be dependent or independent of one another. A subgoal is dependent if its execution must await the successful execution of another subgoal in the same node. It is independent otherwise. An acyclic partial order expresses such a relationship among subgoals. At any stage of the execution of a node, all those subgoals which are independent may be executed asynchronously. However, goals which are candidates for simultaneous execution must be treated specially if they share unbound variables.

A goal selected for execution must be matched against assertion or procedure heads. There may be several assertion/procedure heads which match the given goal. Any procedure head which matches a goal can potentially lead to the solving of that goal, independent of any other procedure head that may

also match the goal. All matching procedure heads are therefore candidates for asynchronous execution. Furthermore, the user may wish to specify a partial order of execution of procedure bodies, in a similar manner to the partial ordering on subgoals within a node. Thus there is the possibility of specifying that certain alternatives need be explored only if other alternatives have failed.

An assertion or a procedure that matches a goal in a node causes a new node to be generated as a successor node to the node that contains the goal. The new node consists of all goals in the parent node where the selected goal is deleted and replaced by the body associated with the procedure head and the matching substitution is applied to the new node. In case of a matching assertion, the body is empty and the new goal node has one less problem to be solved. When an empty node is generated, the problem has been solved.

Executing a problem as outlined above leads to the generation of many nodes, each node of which can be in a partial state of execution. It is in a partial state when all assertion/procedure heads that match a subgoal have not been selected for execution. Thus, there is the option to select many nodes for asynchronous execution.

All possible asynchronous operations may be executed on autonomous machines.

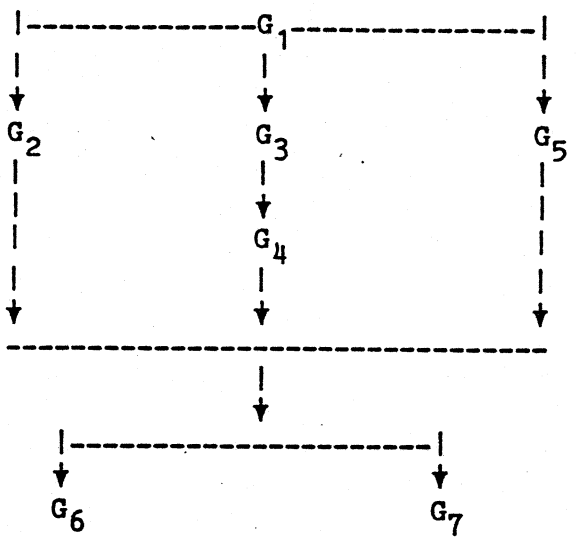
2.3. PRISM Control Facilities and Language

In the previous section we described the possibilities for parallelism in the control structure. Here we specify the support for controlling parallelism in PRISM. PRISM provides the ability to specify for every goal and procedure body a partial order for execution. This partial order expresses the dependencies among the subgoals within a goal (a procedure body may be considered to be a goal). or within alternative procedures for solving the same goal.

The partial order on subgoals in a goal are specified by a notation as explained in the following example.

$$P \leftarrow (G_1,[G_2,(G_3,G_4),G_5],[G_6,G_7]).$$

The procedure head is on the left hand side of the arrow, while the body is the right hand side. The body consists of a set of goals, separated by commas and formed into groups by properly nested pairs of parentheses and brackets. All groups of goals enclosed by parentheses, must be executed in a left-to-right sequence, i.e., the leftmost group in the sequence must be executed and solved before the remaining groups. Groups of goals enclosed in brackets, may be executed independently of other groups in the same set of brackets, i.e., all groups in the bracket may be executed asynchronously. The partial order induced by the above notation is:



The groups formed by G_1; $[G_2,(G_3,G_4),G_5]$; $[G_6,G_7]$, must be executed from left to right since they are enclosed within parentheses, i.e. G_1 must be executed and completed before any other group. Once G_1 is completed, the groups G_2; (G_3,G_4); G_5 may be executed asynchronously since they are enclosed by a bracket. However, since G_3; G_4 are enclosed in parentheses, G_3 must be executed and completed before G_4 is initiated. The next group, $[G_6,G_7]$ cannot be initiated until all groups to its left have been completed, i.e., goals G_1,G_2,G_3,G_4 and G_5. The goals G_6; G_7 may be executed asynchronously.

In the case where no parenthesis or brackets are specified, PRISM assumes a default ordering. This default is user specifiable to be either left-to-right or asynchronous.

The user has the ability to specify a partial-like ordering of procedures with the same procedure name. The user is provided with a notation which permits assigning precedences to procedures. The semantics of the ordering is different than for the ordering of goals. The ordering specified on the procedures is a recommendation on the likelihood of success when the procedure is executed. However, these recommendations may be ignored by the problem solving system which could change the recommended ordering or perform them in parallel. Also provided is a capability to invoke a procedure only if other procedures have been executed and failed. The following notation is used as an example:

1: P <-- G_1,G_2

1: P <-- G_3

2: P <-- G_4,G_5,G_6

*3: P <-- G_7

4: P <-- G_8,G_9

The integers represent the recommended order of execution. The asterisked integers represent a forced ordering. In the above example, the first two procedures (priority=1) may be executed simultaneously. The third procedure (priority=2) is less likely to succeed but may also be executed in parallel with the first two, or even before them if the problem solver so decides. However, the fourth procedure (priority=*3) cannot be executed unless the preceding procedures have been completely executed. A default ordering is provided by PRISM when the procedures are not numbered. The recommended ordering is the sequence in which the procedures were presented to the system.

At the present time, no user facilities are provided for node selection. However, the PRISM problem solver is supplied with several evaluation functions to permit automatic selection of nodes to be expanded.

3. The Problem Solving Machine (PSM)

3.1. The PSM Organization

3.1.1. The Role of the PSM

The Problem Solving Machine (PSM) is the core of the parallel problem solving system. At initiation time, a number of moblets (a moblet is a single ZMOB processing element) are designated as PSMs. The central task of the PSMs is to manage the search space. The complete separation of logic (the problem specification) and control (the strategy of solving the problem) allows a great degree of flexibility while executing the program. Not only can the search strategy be varied dynamically, but due to the inherently nondeterministic nature of logic programs, several mutually exclusive possibilities may be explored simultaneously. The PSMs permit this inherent parallelism to be exploited during the course of solving a problem.

Initially a goal, which represents the problem to be solved, is sent by the VAX to ZMOB and is read by some PSM. This PSM places the goal as the root of a proof tree. A goal is expanded by selecting an atom in the goal and replacing it by the body of a program clause that resolves with it. In this manner a new goal clause, which when solved, solves the original problem, is produced. When an atom is expanded, there may be several program clauses which resolve with it. These represent alternative ways to solve the same problem. These alternative subgoals lead to a branching in the search tree (OR branches).

Thus at any given instant in the problem solution process the search space administered by each PSM consists of a tree of goal clauses. The rest of the search tree is the original goal with which the PSM was initiated. The successor of any clause in this tree is the resolvent obtained by resolving program clauses with some atom in the parent clause. Each leaf node in the tree can be in one of four states:

- the node represents the empty clause
- the node represents a failure node

- the node represents an open goal clause not yet selected for expansion.
- the node represents an active goal clause selected for expansion, but not yet fully expanded.

At any stage the PSM must select an open clause from the search tree, and then select one or more atoms from this clause. This selected atom is then sent to an IDB and/or an EDB for expansion. While the IDB and/or EDB are working on this atom, the PSM can transfer its attention to other nodes in the search tree. An atom sent to the IDB may unify with one or more procedure heads, and all the corresponding bodies are sent back to the PSM which initiated the search, either one at a time or all at once. In the case that more than one procedure body is returned for a given atom, several mutually exclusive subgoal clauses are generated. These mutually exclusive goals can then be solved independently in separate machines.

Thus each PSM has the capability to dynamically send a goal to another PSM machine, if one is available. As with the goal transmitted by the VAX to a PSM, the goal transmitted from one PSM to another becomes a root of a goal tree in the new PSM whose parent is the sending PSM. Each PSM can independently develop and manage the subtrees of the search space generated by the goal node transmitted to the PSM. Each PSM is autonomous except for the knowledge of the parent-child relationship. When a goal assigned to a PSM is completely solved it transmits the solution or failure to its parent PSM. The parent of the PSM to which the original goal was transmitted is the Host (VAX) machine.

3.1.2. Conceptual View of the PSM

This section presents a conceptual view of the program that drives the PSM in terms of the subtasks that compose it and their functional specifications.

The program which drives each PSM is composed of several subtasks. Each subtask operates independently of all other subtasks. These processes do not communicate directly with each other, instead they change global data structures which then may affect another process. This independence makes it possible to consider each process in isolation. This isolation makes the implementation less error prone, and at the same time permits the single PSM process to be split across several machines if the need arises.

The operation of these processes is controlled by a scheduler process which, based on the current state of the global data structures, determines which subtask to invoke next. Thus each process once invoked is allowed to proceed until completion (except in certain special cases, which result in its preemption). Once this process completes, it returns to the scheduler which then applies a decision process to determine which subtask to invoke next. This can be represented by a recursive PROLOG program, of the form:

$$S \leftarrow D_i, P_i, S$$

where S is the scheduler and D_i is a decision process which succeeds if process P_i is to be invoked next, and fails otherwise.

There are six basic processes which compose the problem solver (aside from the scheduler). These are the initialization, input, selection, resolution, output and finalization processes.

The scheduler, once invoked with the initial goal, unconditionally invokes the initialization process which creates all global data structures required by the PSM processes and sets them to their initial values. The scheduler then repeatedly invokes the input, output, selection and resolution processes, by using its decision criteria. This continues until an answer is found or a termination signal is received. Once an answer (or all answers, as the case may be) is found, the finalization process is invoked. If all children PSMs of this PSM have completed already and returned their answers, this PSM transmits its answer to its parent. Otherwise, the finalization process creates a data structure which contains enough information to construct the answers when the children PSMs complete their tasks. Once this data structure is created the PSM is reinitialized and can accept queries.

In this manner PSMs are not kept idle in case they complete before their children do. This also allows a PSM to be its own ancestor if so desired. Thus a cyclic graph of parent-child dependencies may be constructed.

In addition to the six processes mentioned earlier, there are two low level processes which are totally independent of the scheduler and all other processes. These are the mailstop handlers. These processes are interrupt driven and are invoked whenever a message enters (leaves) the input (output) mailstop of the PSM. The input (output) mailstop handler merely places (removes) a message into (from) the input (output) queue and returns to the interrupted process.

The input process understands the message formats of all possible messages that can be received by the PSM. It selects a message from the input queue, decodes it and updates the appropriate global data structure with the information contained in the message.

The output process understands the message formats of all messages that can be sent by the PSM. When invoked with a certain message type, it uses information from the appropriate data structure, and formats this information into the correct message format. This message is then placed into the output queue, ready to be sent out.

The selection process directs the problem solving process by determining which clause, and which atom within the selected clause to operate on next. It is also responsible for the creation of new PSMs.

The resolution process receives the procedure bodies for an atom that has been matched by the IDB and/or EDB. It then inserts a new clause into the proof tree. This new clause consists of the clause from which the atom was selected, with the atom deleted and the procedure body attached in its place. The unifying substitution is then applied to the new clause.

3.2. Control in the PSM

3.2.1. Control Specification Support - Selection Process

The selection procedure determines the control strategy of the system. The user is permitted to specify certain guidelines to direct the selection process. The selection procedure has four main selection functions. These

are: node (clause) selection, atom selection, procedure selection and PSM creation.

Node selection is concerned with choosing a clause, from the search tree, from which an atom is to be selected. Any node which has not been fully expanded, is a candidate for selection. A fully expanded node consists of a clause whose selected atom has been expanded and all leaf nodes descended from the clause are either failure nodes or null clauses. A non-fully expanded node may be either an active or an open node. An active node is one from which one or more atoms have been selected for expansion, but which has not been fully expanded. An open node is one from which no atom has yet been selected for expansion.

Atom selection is concerned with selecting an atom, for expansion, from a selected node in the search tree. There are several system defined and user defined constraints that will affect atom selection.

As defined in section 2, the user has the ability to specify which atoms in a clause may be executed in parallel and which must be done in sequence, i.e. a partial order on the execution of the atoms. These user specified constraints limit the atoms which can be selected at any stage. Only those atoms which do not depend on any other atom or those for which the atoms they depend on have already been solved are candidates for selection.

In addition to these user-defined orderings, there are certain orderings implied by the structure of the node itself. There are two basic ways in which the contents of a clause dictate the ordering on atom selection. These are: dependent atoms, and special predicates.

Two or more atoms in a clause are said to be dependent when they share variables. In this case what is desired is the first (or all) binding(s) which cause the atoms to succeed. This can be accomplished either by processing the atoms in parallel and then intersecting the sets of bindings for the shared variables, or by finding a binding which satisfies one predicate and then substituting it in the others and determining if they succeed with that binding. This can be repeated until some binding succeeds or all are exhausted (nested loops method). In either method a special AND node has to be generated with the dependent atoms as its children and one of the above techniques applied. In this system the nested loops method will be adopted since space limitations make the set of values method infeasible.

The special predicates are a set of language supplied predicates whose semantics dictate that certain other predicates in the clause must be fully solved before these system defined predicates may be invoked, i.e., these predicates induce a partial ordering on the atoms in a clause. These predicates are: write, read, fail, / (the cut operator), not, and the evaluable predicates (e.g., arithmetic operations).

Once these user and system defined constraints have been satisfied, a set of atoms which are candidates for selection will remain. The atom selected from this set will be selected based on user or system supplied heuristics.

Procedure selection is concerned with choosing which procedure body should be given the highest priority when several bodies match an atom which was sent for expansion. This decision is made by the IDB and is not influenced by the PSM.

PSM selection, is concerned with the decision of when to initiate another PSM with a subproblem. Whenever a branching of the search tree is induced by either multiple alternate subproblems (OR-branches) or by independent conjunctive subproblems (AND-branches), this branch becomes a candidate for execution in another PSM machine. The actual process of determining when a new PSM is initiated is discussed in the following section.

3.2.2. PSM Creation

The decision of when to initiate another PSM with a subproblem is not an easy one. If the subproblem is too small, a large amount of overhead would be incurred to solve it. If the subproblem is too large, the parent PSM may complete before the child and remain idle until its children complete. In general it is extremely difficult if not impossible to determine how complex a subproblem is. Thus no attempt is made to determine the complexity of a subproblem, in the initial system. Instead a new PSM is initiated every time a branching of the search tree takes place, and there is a PSM available. At any given instant, there may be several active branches within a PSM, and thus several candidate nodes which may be sent to other machines. In this case all or only some of these nodes may be shipped out. This is determined by the user or by system supplied heuristics.

In order to reduce idle time, machines which have completed their alloted task are permitted to accept fresh queries, as follows. If no further processing can be done then either all possible answers for the goals this PSM was invoked with have been found, or all paths resulted in failure, or all paths local to this PSM have been fully explored and there are some children of this PSM which have not yet completed their work. In the cases where all answers have been found or all paths have failed, this information is transmitted to the parent of this PSM, and the PSM state is restored to one in which a new query can be accepted. In the case where all local paths have been explored and some chidren PSMs are still active, a data structure is constructed which contains enough information to reconstruct the complete answer from the information in this PSM and from the answers from the currently active children PSM. Once this data structure is constructed, the local proof tree is destroyed and the PSM state is restored to one in which a new query can be accepted.

In this manner PSMs are not kept idle in case they complete before their children do. This also allows a PSM to be its own ancestor if so desired. Thus a cyclic graph of parent-child dependencies may be constructed.

3.2.2.1. AND Parallelism

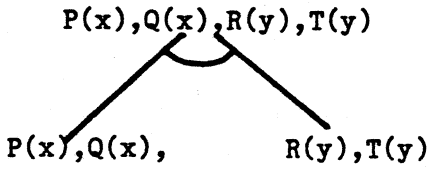
An AND-branch in the search tree can be one of two types. The first type of AND-branch results when there is a conjunction of atoms which do not share variables. This results in a node which has as its children two or more sets of atoms which do not share variables. We shall refer to such an AND-node as an AND_1-node. The second type of AND-branch results when there is a conjunction of atoms which do share variables (dependent atoms). This results in a node which has as its children two or more sets of atoms which do share variables. These children are ordered so that those atoms which bind variables are executed earlier than those atoms which use those variables. Such an

AND-node will be referred to as an AND_2-node.

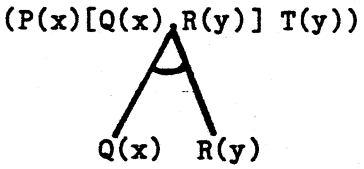
The children of an AND_2-node are currently always executed in the same machine since concurrent execution across machines requires an excessive amount of control and communication to synchronize the producers and consumers of the answers. Thus in the initial system the children of only AND_1-nodes are executed concurrently in separate machines.

We previously defined a clause to be an AND_1-Node if it could be split into two or more sets of atoms that do not share variables.

This definition must be revised when dealing with ordered clauses. For example let <- P(x),Q(x),R(y),T(y) be a clause. According to the definition above we create an AND_1-Node as follows:



Now let <-(P(x),[Q(x),R(y)],T(y)) be an ordered clause. The execution sequence imposed by the order in the clause does not allow a similar split. Thus a split may be performed only on sets of the clause that may be executed in parallel. Therefore the clause above is represented as:



The split would be executed dynamically after P was solved.

3.2.2.2. OR Parallelism

An OR-branch is created in the search tree when there are several matching bodies for a selected atom. These several bodies are alternative ways of solving the selected subgoal and are thus independent. The existence of multiple bodies then results in the formation of an OR-node with each of these bodies as a child.

Since these children are independent of each other they may be executed in separate machines. However those bodies which should not be attempted until some other body fails are not scheduled for execution until the body it depends on has failed.

3.2.3. Handling Negation and the CUT(/) Operator

In addition to imposing an implicit ordering on the atoms within a node, the cut(/) operator and negation both require special treatment in concurrent systems.

3.2.3.1. The CUT(/) Operator

The cut operator is a means of achieving determinism in sequential logic programs. The execution of a cut in the body of a procedure results in all alternatives for the parent node of the node containing the cut to be discarded. However, in sequential execution all alternatives with higher priority than the one containing the cut have already been completely executed before the one containing the cut. Thus the semantics of the cut operator are unclear for concurrent execution, since the alternative containing the cut may be executed concurrently with, or even before alternatives with higher priority. This would lead to an incompatibility between sequential and concurrent execution of the same logic program.

In the interest of preserving compatibility between the concurrent and sequential execution of logic programs containing the cut operator, we have defined the concurrent cut as detailed below.

The presence of the cut operator causes an implicit ordering of atoms within the node containing the cut. The cut operator requires all atoms preceding it in the node to be processed before it. The invocation of this operator results in the following:

(a) all bindings that have been computed for variables in atoms preceding the cut will not be recomputed in the event of a failure of some atom succeeding the cut

(b) all alternative procedure bodies which have lower priority than the body containing the slash are discarded. However all higher priority bodies are still considered, i.e., if procedure P has 3 bodies and if the second (middle priority) body contains a cut then the third body will never be considered if the cut in the second is executed, but the first body will be unaffected.

3.2.3.2. Negation

The NOT meta-predicate defined in most sequential logic programming is an implementation of Negation-by-Failure [Clark 1978]. In this scheme, the negation of an atom is considered to hold if all attempts to prove the atom, fail. This is well defined in the case where all the arguments of the atom are constants. However this is not well defined when one or more of the arguments are unbound variables. This is because the atom could succeed with some particular bindings for the free variables, and fail for some other bindings. Thus the behaviour of the NOT meta-predicate can be anomalous in the case where all argumentds are not constants. The semantics of negation can be extended to handle atoms with variables as arguments by creating a set of bindings for which the atom fails and assuming the negation of the atom holds for precisely this set of bindings. This is how we define negation in PRISM.

The NOT meta-predicate requires that all atoms preceding it in the node must have been solved before it is invoked. The execution of this meta-predicate results in the creation of a special negation node which has as children the predicates which are to be negated. These predicates are solved as if they were positive atoms for whom all answers are desired. When all these predicates have been solved, the sets of values bound to each variable will be complemented with the domain over which they are defined. These

complements will be returned as answers by the negation node.

4. Examples of Control in PRISM

In this section we provide an example of AND-parallelism and an example of OR-parallelism to illustrate some of the capabilities of PRISM.

4.1. AND Parallelism

This example of AND-parallelism provides a preorder traversal of a binary tree.

Let

P(u,z) means that the preorder traversal of a binary tree u is z.
$t(y_1,x,y_2)$ denote a tree whose left branch is y_1, whose root is x and whose right branch is y_2.
$Append(y_1,y_2,z)$ mean that if y_2 is appended to the tail of y_1 the result is z.

We may then write,

1: P(nil,nil) <--

*2: $P(t(y_1,x,y_2),x.z)$ <--
$([P(y_1,z_1),P(y_2,z_2)],Append(z_1,z_2,z))$.

1: Append (nil,y,y) <--

*2: Append (x.y,v,x.w) <-- Append (y,v,w).

Thus, the base case, P(nil,nil) is always tested before the general case. When the preorder clause defined by *2 is executed, the preorder traversal of the left and right branches may be done asynchronously.

Since the preorder traversal of the left branch is independent of the right branch, they may be executed asynchronously in different processors. Each of the sub-branches may again be split to be executed on different machines. Hence, a number of different PSMs can be executing the problem at the same time. Thus, the time to execute the search is proportional to the size of the longest branch, rather than the number of nodes in the tree as in a sequential search.

Even if the problem is executed on a single machine, because one can be searching for matches on many nodes of the search tree, the multiple IDB machines can be working in parallel performing matching operations for each of the nodes in the search tree.

4.2. OR Parallelism

Consider a database problem whose database is shown below (Figure 2).

Extensional (Relational) Database

```
MOTHER(Rita, Sally)
MOTHER(Alice, Beth)
MOTHER(Laura, Christine)

FATHER(Harry,Jack)
FATHER(Harry, George)
FATHER(Jack, Sally)
```

Intensional Database

```
GRANDPARENT(x,z)   <- [Parent(x,y),Parent(y,z)]
PARENT(u,z)        <-  Mother(u,z)
PARENT(u,z)        <-  Father(u,z)

Query:   <- GRANDPARENT(x, Sally).
```

Figure 2.

We shall describe how the problem might be solved in PRISM on the ZMOB system. We use the following abbreviations in the series of figures that follow.

M-MOTHER	R-RITA	H-HARRY
F-FATHER	S-SALLY	G-GEORGE
G-GRANDPARENT	A-ALICE	J-JACK
	B-BETH	

We assume that there are only two moblets assigned to the EDB, one moblet contains the F-table, and the other the M-table. We assume that the IDB is replicated on two moblets and two moblets are allocated to be PSMs.

When the system is to be loaded, the ZMOB executive specifies the moblets allocated to the problem. The PRISM executive is informed of the machines and allocates the EDB, IDB, and PSMs to specific moblets. The data and programs are sent in a burst mode by the PRISM executive, resident in the host machine, to the appropriate moblets. Mask registers are set in the EDB moblets so that they can recognize the encoding for MOTHER and FATHER. The state of the system as would exist on ZMOB is illustrated in Figure 3. Processing of the query shown in Figure 2 is now described.

1. The query is submitted by the user to the PRISM executive which sends a message requesting response from a free PSM. We assume that the PSM on the first moblet encountered responds and that the query is sent over the belt in a burst mode. Figure 4 shows the state of the system at this point.

2. The PSM-1 forms a goal tree, and selects the only atom to be solved. It knows from preprocessing that the "G" predicate resides in an IDB machine. It sends it out to be matched against all procedure heads with

the same name. IDB-1 receives the request and also notes that there is only one response possible. See Figure 5.

3. IDB-1 finds a single match, informs PSM-1 that it has a match and at the request of PSM-1 transmits the body of the procedure and the unifying substitution. See Figure 6.

4. PSM-1 forms a new node (the resolvent clause) and selects the easier of the two subproblems to be solved, namely PARENT(y,S). It determines that the PARENT relation is intensional, and sends a message out to an IDB to be processed. Since IDB-1 is not busy, it accepts the message and now finds two matches. The PSM-1, in the meantime, determines that it has no work to be done since no additional responses are possible for the root node and it must wait for a response.

5. PSM-1 is informed by IDB-1 when it has found all matches for PARENT(y,S). There are two responses. PSM-1 may request that both responses be transmitted (or it may be done one at a time). Assuming both are transmitted, the PSM forms two nodes (OR branches). Since a PSM is available, it transmits one of the two nodes to PSM-2 to be solved. Now, PSM-1 can send out a request for MOTHER(y,S), while PSM-2 can send out a request for FATHER(y,S). These requests are sent out by pattern on the relation name and accepted by different moblets where the two relations are stored.

At this stage, two PSMs are cooperating in the solution of the problem, and two EDB machines are searching for data. The processing continues in a manner similar to the above description, until a solution is arrived at, as shown in Figure 3-9.

The above illustrates how OR parallelism may be handled within PRISM. Both AND and OR parallelism may be executing simultaneously. Each of the PSM machines may be working on a problem at some stage and all IDB and EDB machines may also be executing simultaneously.

5. Summary and Future Work

There have been several proposals to achieve parallelism in logic programming systems (Clark[1981], Hogger[1982], Pereira[1978], van Emden[1976], Wise[1982]). All these schemes, including PRISM provide natural ways of expressing algorithms for execution on conventional distributed architectures.

PRISM provides an implementation of logic programs on a highly parallel architecture. The design exhibits a high degree of modularity and orthogonality. By this we mean that portions of PRISM can be replaced by functionally similar modules with a minimum impact on the system. This provides a flexible tool to experiment with the implementation of various control strategies on diverse architectures. It provides full OR parallelism, partial AND parallelism and permits parallel asynchronous search for assertions and procedures. Parallelism is transparent to the user. We provide a proper interpretation of negation based on negation-by-failure.

The system represents a first approach to developing an experimental tool for the design and implementation of large AI systems. There are many capabilities that need to be added in a second development. Some of these are: co- routining; a full AND-parallelism; user specificable heuristics; typing of

variables; intelligent backtracking for arbitrary execution sequences; non-top-down search methods; and the ability to incorporate lemmas dynamically. These capabilities need to be incorporated into a coherent control language that would permit the user to specify diverse aspects of control to varying depths of detail while a problem is being solved. Some of the issues related to the above developments are explored below.

The fundamental difficulty in distributed problem solving arises from the fact that distributed control has not been cohesively studied and it is hard to achieve effective global solution by distribution of tasks: good local decisions are not necessarily a guarantee for an effective global procedure. Thus our efforts were aimed at the construction of a system that will be able to support various problem solving strategies without paying the price in efficiency of the execution. The main emphasis in our system was directed towards modularity, flexibility and adaptivity. We believe that a paper design is rarely as good as an effective implementation on a real parallel machine, which will enable modifications and enhancements with minimal programming effort, and consequently were admittedly willing to make some compromises in the initial system. Consistent with this philosophy the system components induce a logical network topology, and virtual processors may be added or deleted easily in our system without any changes to the rest of the system. Each set of machines is seen as a single machine to the rest of the system and any modifications and improvements to individual components may be made without effecting the rest of the system. In this section we briefly discuss some of the enhancements to PRISM that are currently under implementation or being investigated.

Database Machines

EDB - Today's databases are far larger than the memory capacity of a few hundred 64K microcomputers. Thus it will be useful to incorporate in our system a set of peripheral devices that will be attached to each EDB(or possibly shared by several EDBs). This will enable both an increased memory capacity and an ability to dynamically reconfigure the database machines in case of an unbalanced demand on one of the EDB machines. Additionally it will be constructive to facilitate basic database operators such as join,projection for efficient data retrieval.

IDB - In the current implementation the set of intensional database axioms (IDB) is replicated over several machines. This philosophy was based on the assumption that the IDB is relatively small and therefore may be stored in one machine. This assumption simplified the communication protocols and the operation of each IDB machine. We are currently developing a scheme in which the IDB is distributed over several machines. Additionaly to achieve effective performance from a highly parallel machine there is a need to control the ratio between the communication and the computation time, that is for a full utilization of a system it is desirable to increase the computation and minimize communication. In the system to date this control exists in the PSMs which may decide to solve a subproblem themselves rather than dispatch it to a different PSM. Similar techniques will be incorporated in the IDB. The IDB machine could perform several atomic resolution steps before returning the bodies and the unifiers to the PSM, thus increasing the ratio between computation and communication involved in a single resolution step. This effect may be attained either by a partial compilation of the the IDB axioms, or by

parameterized execution of the IDB that will perform a number of resolution steps as indicated by the PSM that initiated the query.

PSM - Machines Structure Sharing

Currently there is no sophisticated memory management done in the PSM. The memory management schemes most commonly used in Prolog implementations are copying and structure sharing. The decision not to incorporate structure sharing in PRISM was motivated by two factors. Firstly an implementation of a straight forward structure sharing will result in a tremendous overhead in pointer chaining before each literal(clause) is sent for expansion or a cumbersome and inefficient resolution operation if the structure sharing is done across processors. Secondly, since more than one path in the proof tree is active during execution, a locking mechanism must be incorporated to disallow bindings from two different paths to be applied to variables of the same literal simultaneously.

Parallelism enhancements in the PSM.

The flexible implementation of the selection procedures allows some dynamic exploitation of inherent parallelism in the program. This includes automatic detection of literals that do not share variables,and selecting literals that will maximize the degree of the parallelism in the new subgoal. Consider the goal : <- P(x),Q(x,y),R(y,z). It is quite clear that if Q is an EDB predicate, binding of x and y in Q will result in a new goal with two literals that do not share variables, and therefore maximize the parallelism in the clause. At the moment our system supports only local detection of parallelism, that is parallelism internal to a single clause. We are currently investigating partial compilation techniques to maximize global parallelism, during execution, and global planning strategies to optimize the performance of the system in terms of utilization of the computational power of ZMOB on the one hand and search space pruning on the other.

It is evident that cooperative problem solving must be supported with communication channels between PSMs to minimize some of the redundant search, by sharing partial results, eliminating goals that are logically related (by implication or subsumption) and task sharing.

The notion of a PSM as defined in our system is problem independent,that is each PSM may accept any problem. We are investigating the possibility of an Expert PSM which is dedicated to the solution of a class of problems. This notion will minimize some of the effort spent by the PSM in the selection process by precompiling some or all of the selection procedures.

Our system is based on a goal driven procedure invocation . Some thought has been given to facilitation of data-driven procedural invocation, that will allow effective simulation of data-flow machines.

Before any of the above enhancements are attempted we will need to perform many experiments with PRISM to determine its strengths and weaknesses. We plan to experiment with algorithms by alternatively modifying the logic, the control, and the architecture.

6. Acknowledgements

Work on this effort was supported by AFOSR grant number 82-0303 and by NSF grant number MCS-79-19418.

7. References

Chakravarthy[1982]
Chakravarthy, U.S., Kasif, S., Kohli, M., Minker, J., Cao, D., "Logic Programming on ZMOB: A Highly Parallel Machine", Proc. 1982 International Conference on Parallel Processing, IEEE Press, 1982, New York, pp 347-349.

Clark[1978]
Clark, K.L., "Negation as Failure", in Logic and Databases, H. Gallaire and J. Minker (Eds.), Plenum Press, 1978, New York.

Clark[1981]
Clark, K.L., Gregory, S., "A Relational Language for Parallel Programming", DOC 81/16, Dept. of Computing, Imperial College, 1981, London.

Computer[1982a]
Computer, Vol. 15, No. 2, Special issue on Data Flow Systems, February 1982, IEEE Press, New York.

Computer[1982b]
Computer, Vol. 15, No. 1, Special issue on Highly Parallel Computing, January 1982, IEEE Press, New York.

Eisinger[1982]
Eisinger, N., Kasif, S., Minker, J., "Logic Programming: A Parallel Approach", Technical Report 1128, Dept. of Computer Science, University of Maryland, College Park, 1981.

Hewitt[1977]
Hewitt, C., "Viewing Control Sructures as Patterns of Passing Messages", Artificial Intelligence, Vol. 8, North-Holland Publishing Company, 1977, pp 323-364.

Hogger[1982]
Hogger, C.J., "Concurrent Logic Programming", in Logic Programming, K.L. Clark and S-A. Tarnlund (Eds.), Academic Press, 1982, New York.

Kahn[1977]
Kahn, G., MacQueen, D.B., "Coroutines and Networks of Parallel Processes", Proc. IFIP Congress 77, North Holland, Amsterdam, pp 564-569.

Kornfeld[1979]
Kornfeld, W.A., "Using Parallel Processing for Problem Solving", A.I. Memo No. 561, MIT A.I. Lab, December 1979, Cambridge, MA.

Kowalski[1979]
Kowalski, R.A., "Logic for Problem Solving", Elsevier North Holland Inc., 1979, New York.

Kung[1980]
Kung, H.T., "The Structure of Parallel Algorithms", in Advances in Computers 1980, M.C. Yovits, Ed., Academic Press, 1980, pp 65-112.

Lieberman[1981]
Lieberman, H., "Thinking About Lots of Things At Once Without Getting Confused", A.I. Memo No. 626, MIT A.I. Lab, May 1981, Cambridge, MA.

Minker[1982]
Minker, J., Asper, C., Cao, D., Chakravarthy, U.S., Csoeke-Poeckh, A., Kasif, S., Kohli, M., Piazza, R., "Functional Specification of the ZMOB Parallel Problem Solving System", Technical Note Z-1, Dept. of Computer Science, University of Maryland, College Park, 1982

Pereira[1978]
Pereira, L.M., Monteiro, L.F., "The Semantics of Parallelism and Co-Routining in Logic Programming", Divisao de Informatica, Laboratorio Nacional de Egenhario Civil, December 1978, Lisbon.

Rieger[1980]
Rieger, C., Bane, J., Trigg, R., "ZMOB : A Highly Parallel Multiprocessor", TR-911, Dept. of Computer Science, University of Maryland, May 1980, College Park, Mayland.

Rieger[1981a]
Rieger, C., Trigg, R., Bane, J., "ZMOB : A New Computing Engine for AI", TR-1028, Dept. of Computer Science, University of Maryland, March 1981, College Park, Maryland.

Rieger[1981b]
Rieger, C., "ZMOB : Hardware from a User's Point of View", TR-1042, Dept. of Computer Science, University of Maryland, April 1981, College Park, Maryland.

van Emden[1976]
van Emden, M.H., Lucena, G.H., de Silva, H.M., "Predicate Logic as a Language for Parallel Programming", Research Report, Dept. of Computer Science, Univ. of Waterloo, Ontario.

Wise[1982]
Wise, M.J., "A Parallel Prolog: The Construction of A Data Driven Model", Proc. 1982 ACM Symposium on LISP and Functional Programming, ACM Press, 1982, New York, pp 56-66.

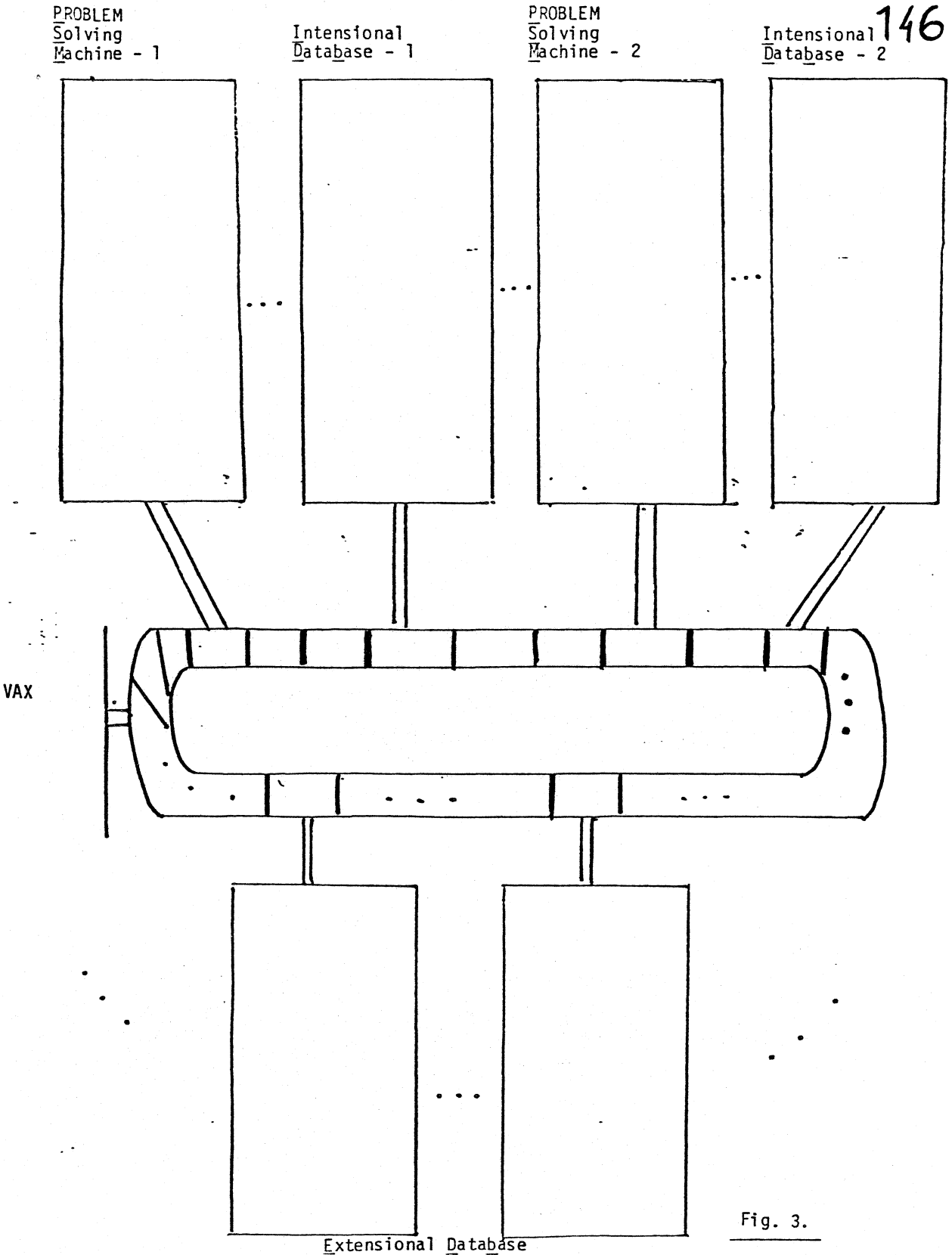

PROBLEM Solving Machine - 1
Intensional Database - 1
PROBLEM Solving Machine - 2
Intensional Database - 2
VAX
Extensional Database

Fig. 3.

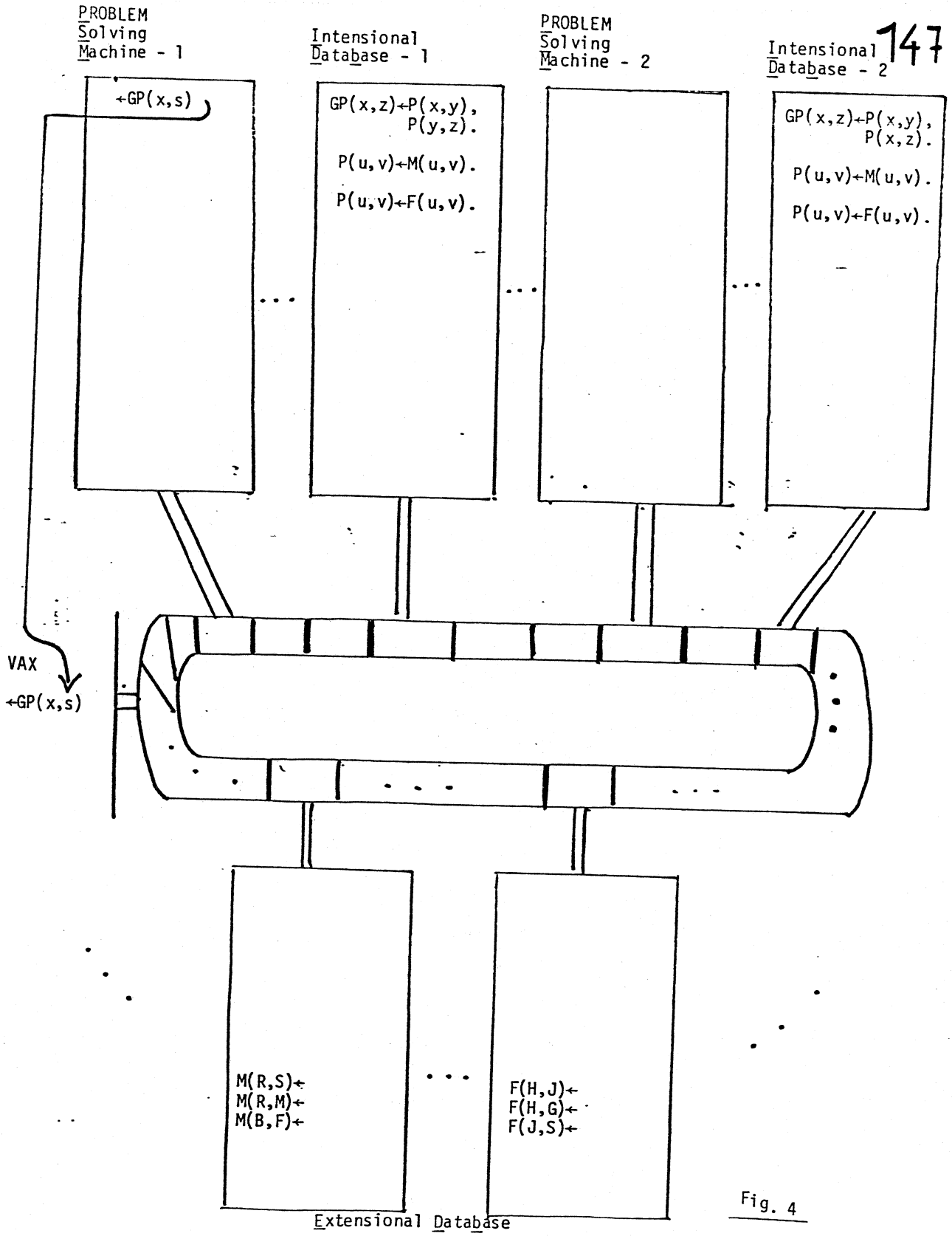

PROBLEM Solving Machine - 1
Intensional Database - 1
PROBLEM Solving Machine - 2
Intensional Database - 2
←GP(x,s)
GP(x,z)←P(x,y), P(y,z).
P(u,v)←M(u,v).
P(u,v)←F(u,v).
GP(x,z)←P(x,y), P(x,z).
P(u,v)←M(u,v).
P(u,v)←F(u,v).
VAX
←GP(x,s)
M(R,S)←
M(R,M)←
M(B,F)←
F(H,J)←
F(H,G)←
F(J,S)←
Extensional Database

Fig. 4

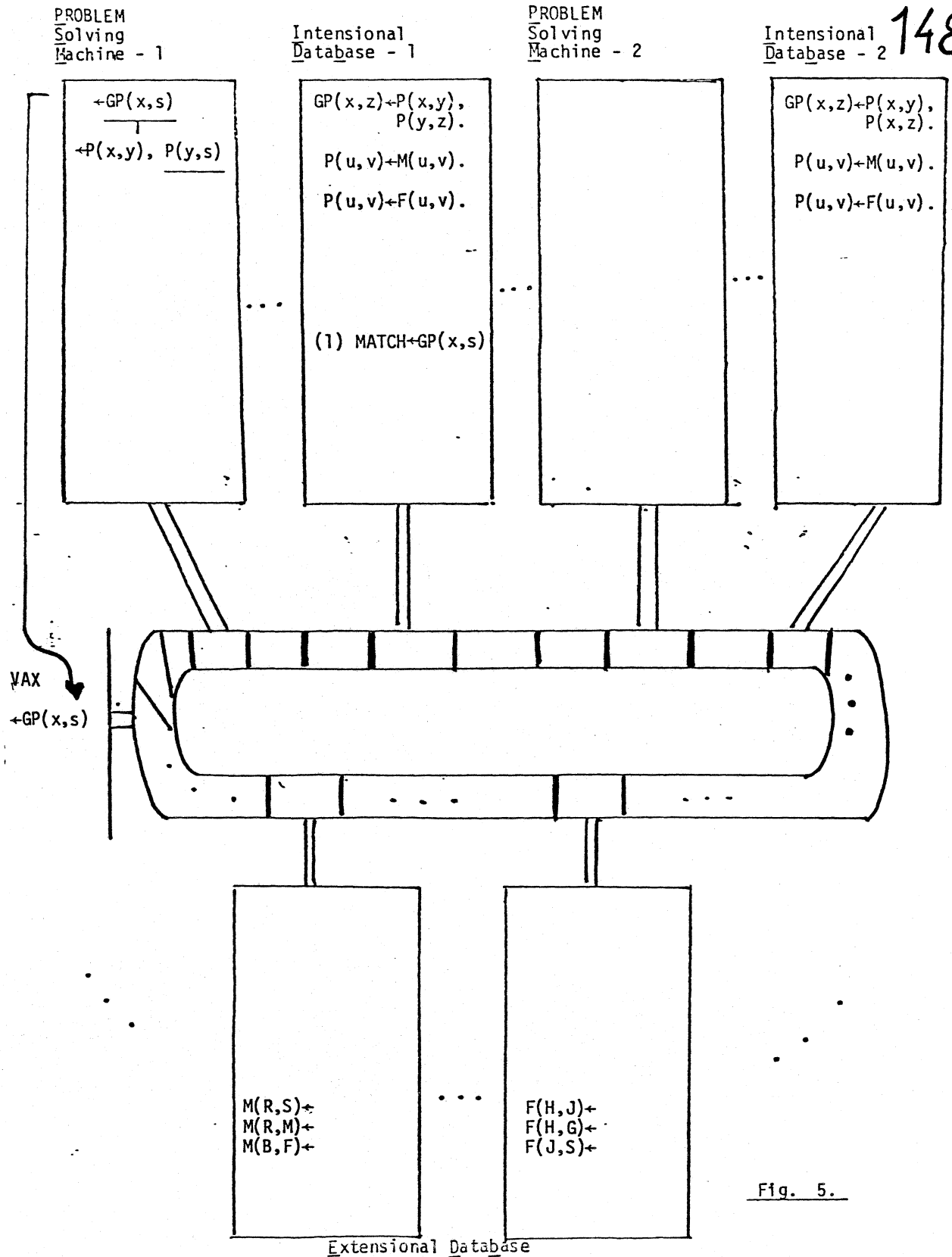

PROBLEM Solving Machine - 1
Intensional Database - 1
PROBLEM Solving Machine - 2
Intensional Database - 2
←GP(x,s)
←P(x,y), P(y,s)
GP(x,z)←P(x,y), P(y,z).
P(u,v)←M(u,v).
P(u,v)←F(u,v).
(1) MATCH←GP(x,s)
GP(x,z)←P(x,y), P(x,z).
P(u,v)←M(u,v).
P(u,v)←F(u,v).
VAX
←GP(x,s)
M(R,S)←
M(R,M)←
M(B,F)←
F(H,J)←
F(H,G)←
F(J,S)←
Extensional Database

Fig. 5.

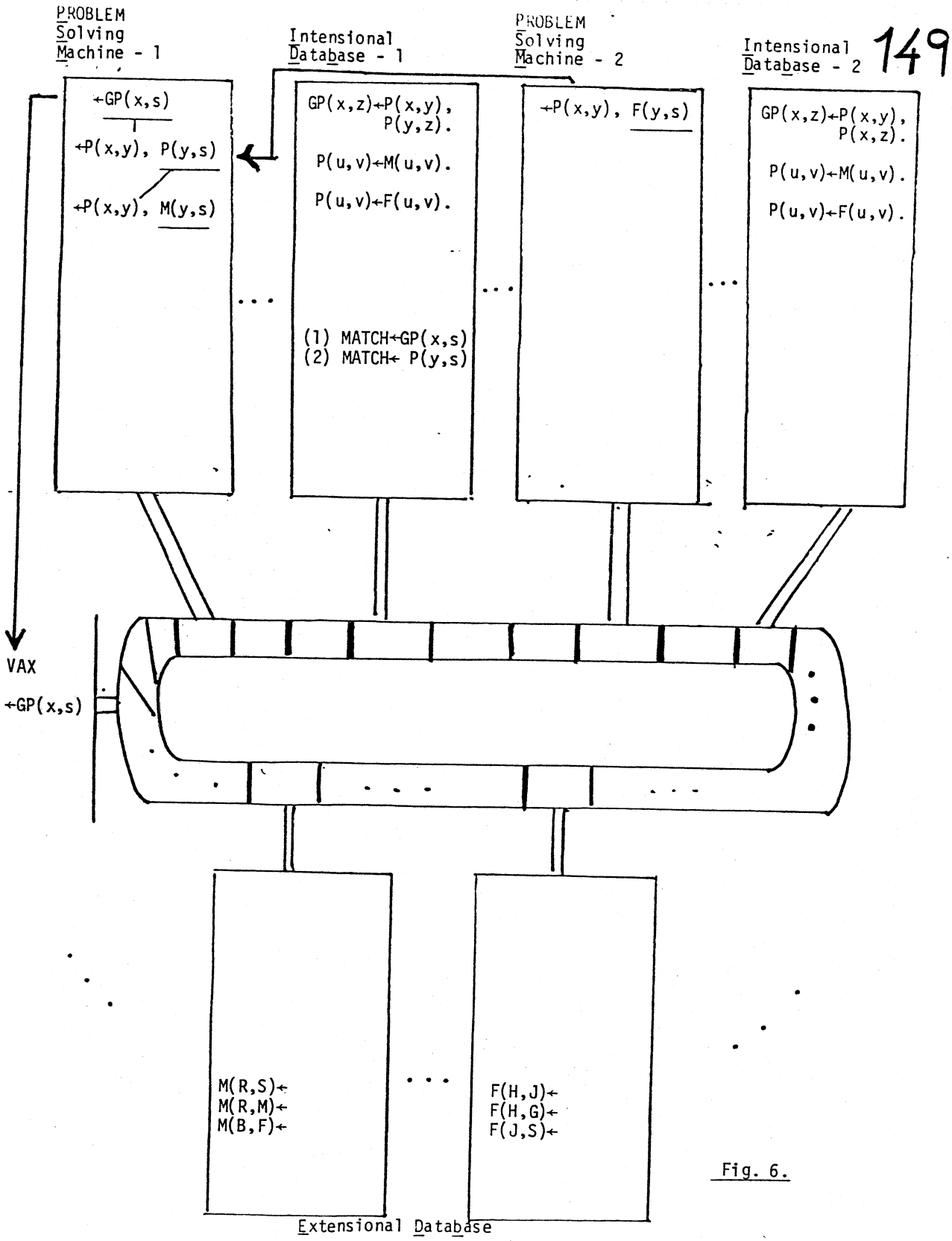

PROBLEM Solving Machine - 1
Intensional Database - 1
PROBLEM Solving Machine - 2
Intensional Database - 2
←GP(x,s)
←P(x,y), P(y,s)
←P(x,y), M(y,s)
GP(x,z)←P(x,y), P(y,z).
P(u,v)←M(u,v).
P(u,v)←F(u,v).
(1) MATCH←GP(x,s)
(2) MATCH← P(y,s)
←P(x,y), F(y,s)
GP(x,z)←P(x,y), P(x,z).
P(u,v)←M(u,v).
P(u,v)←F(u,v).
VAX
←GP(x,s)
M(R,S)←
M(R,M)←
M(B,F)←
F(H,J)←
F(H,G)←
F(J,S)←
Extensional Database

Fig. 6.

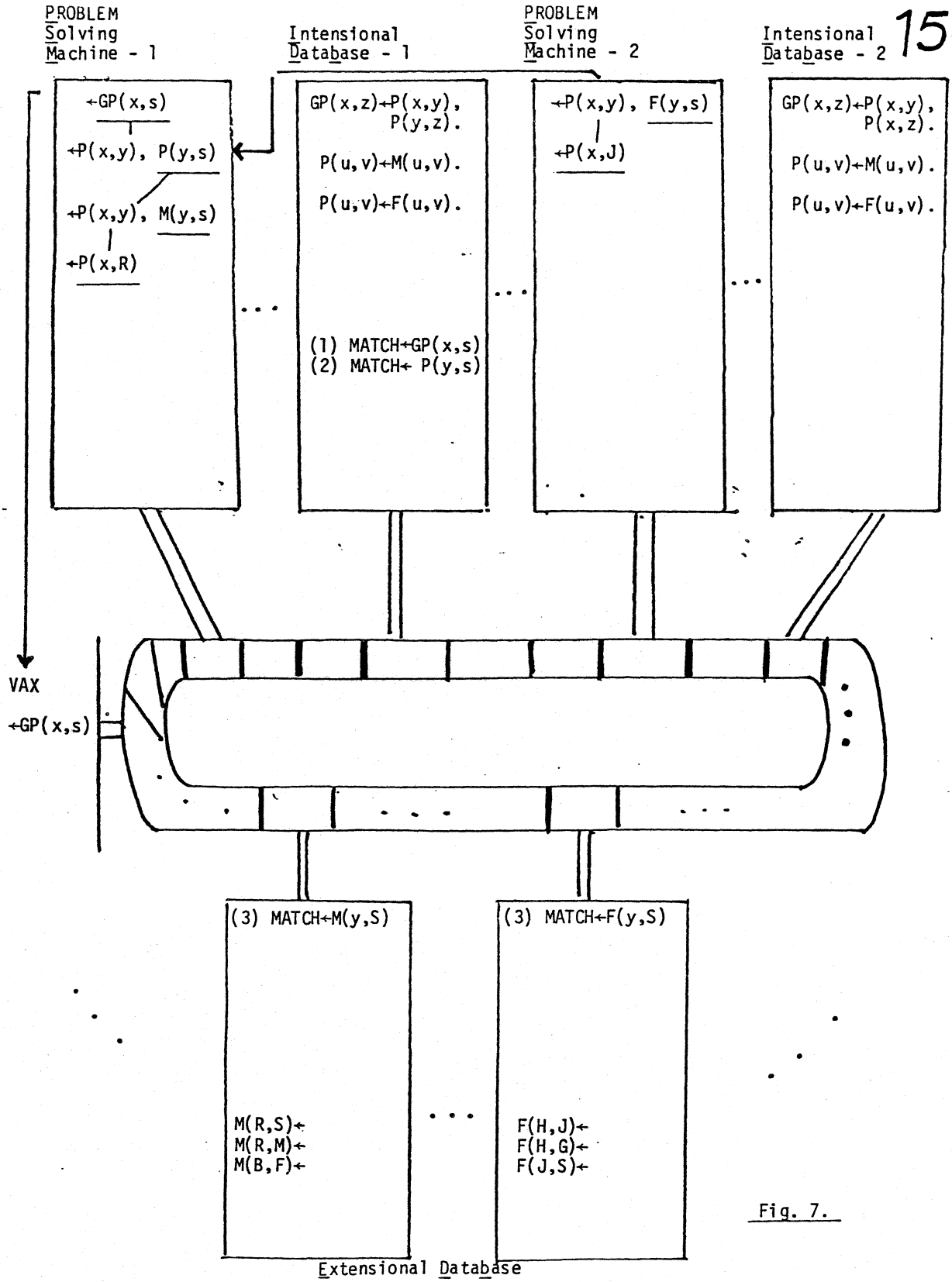

PROBLEM Solving Machine - 1
Intensional Database - 1
PROBLEM Solving Machine - 2
Intensional Database - 2
←GP(x,s)
←P(x,y), P(y,s)
←P(x,y), M(y,s)
←P(x,R)
GP(x,z)←P(x,y), P(y,z).
P(u,v)←M(u,v).
P(u,v)←F(u,v).
(1) MATCH←GP(x,s)
(2) MATCH← P(y,s)
←P(x,y), F(y,s)
←P(x,J)
GP(x,z)←P(x,y), P(x,z).
P(u,v)←M(u,v).
P(u,v)←F(u,v).
VAX
←GP(x,s)
(3) MATCH←M(y,S)
(3) MATCH←F(y,S)
M(R,S)←
M(R,M)←
M(B,F)←
F(H,J)←
F(H,G)←
F(J,S)←
Extensional Database

Fig. 7.

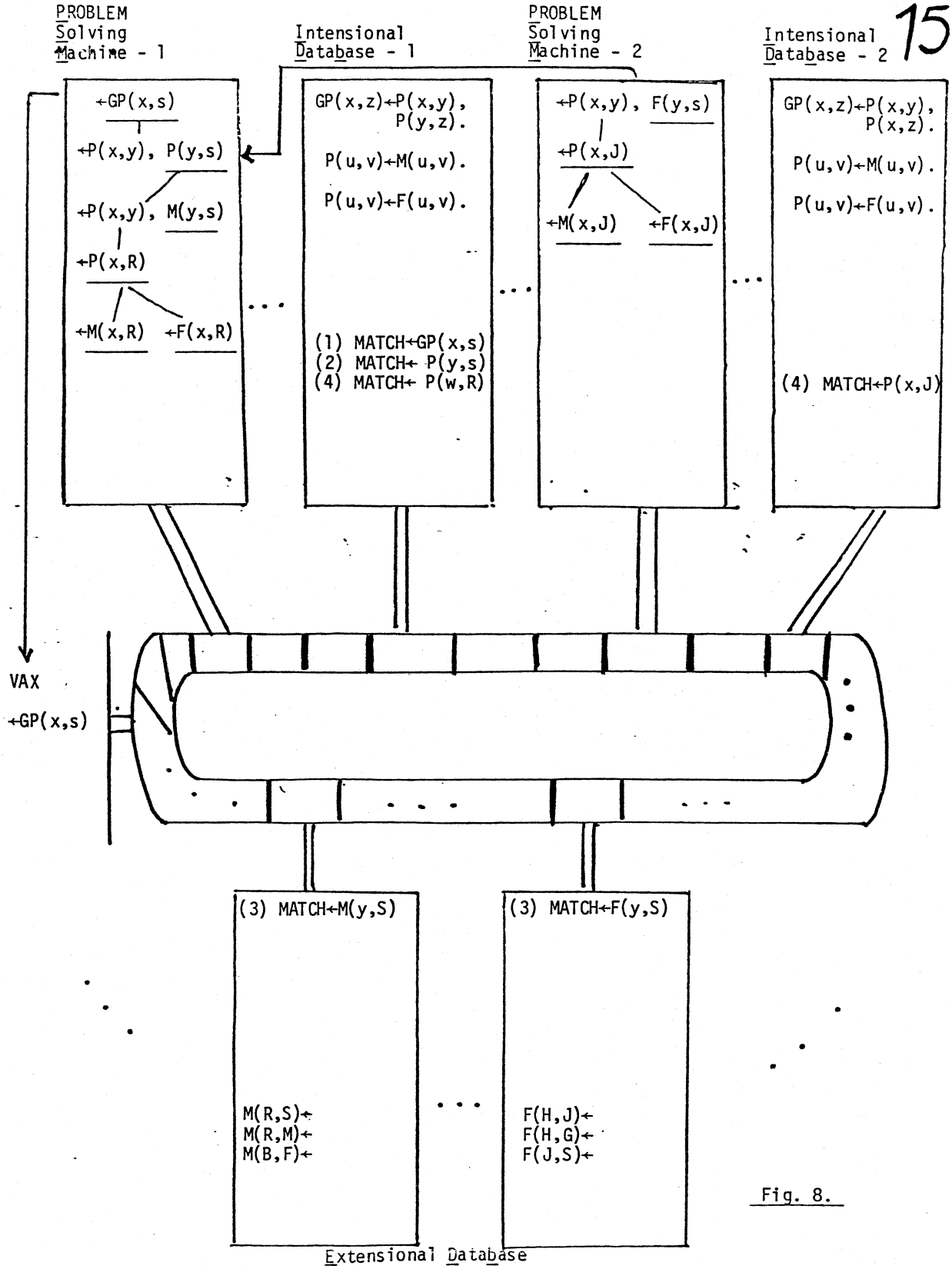

PROBLEM Solving Machine - 1
Intensional Database - 1
PROBLEM Solving Machine - 2
Intensional Database - 2
←GP(x,s)
←P(x,y), P(y,s)
←P(x,y), M(y,s)
←P(x,R)
←M(x,R) ←F(x,R)
GP(x,z)←P(x,y), P(y,z).
P(u,v)←M(u,v).
P(u,v)←F(u,v).
(1) MATCH←GP(x,s)
(2) MATCH← P(y,s)
(4) MATCH← P(w,R)
←P(x,y), F(y,s)
←P(x,J)
←M(x,J) ←F(x,J)
GP(x,z)←P(x,y), P(x,z).
P(u,v)←M(u,v).
P(u,v)←F(u,v).
(4) MATCH←P(x,J)
VAX
←GP(x,s)
(3) MATCH←M(y,S)
(3) MATCH←F(y,S)
M(R,S)←
M(R,M)←
M(B,F)←
F(H,J)←
F(H,G)←
F(J,S)←
Extensional Database

Fig. 8.

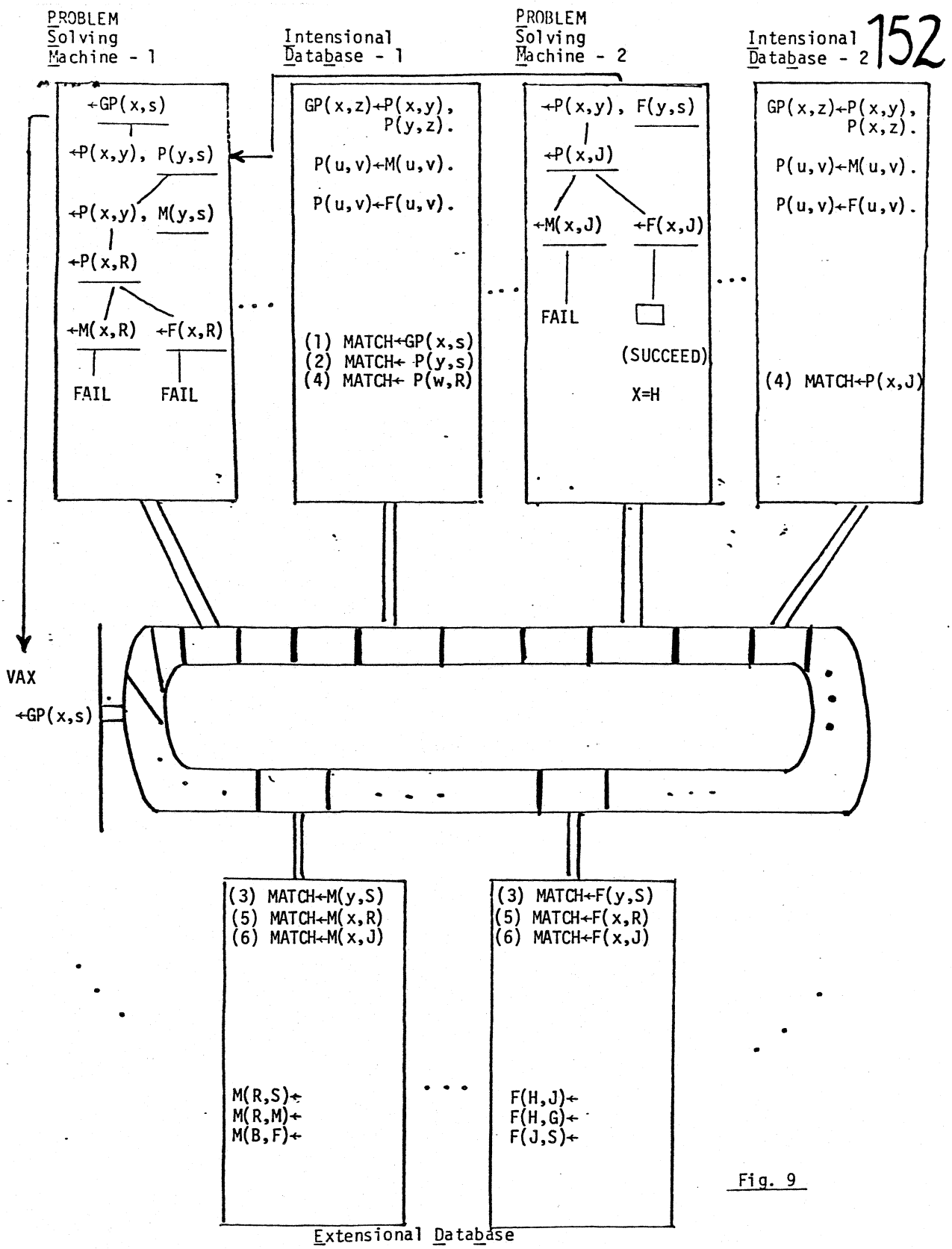

PROBLEM Solving Machine - 1
Intensional Database - 1
PROBLEM Solving Machine - 2
Intensional Database - 2
←GP(x,s)
←P(x,y), P(y,s)
←P(x,y), M(y,s)
←P(x,R)
←M(x,R)
←F(x,R)
FAIL
FAIL
GP(x,z)←P(x,y), P(y,z).
P(u,v)←M(u,v).
P(u,v)←F(u,v).
(1) MATCH←GP(x,s)
(2) MATCH← P(y,s)
(4) MATCH← P(w,R)
←P(x,y), F(y,s)
←P(x,J)
←M(x,J)
←F(x,J)
FAIL
(SUCCEED)
X=H
GP(x,z)←P(x,y), P(x,z).
P(u,v)←M(u,v).
P(u,v)←F(u,v).
(4) MATCH←P(x,J)
VAX
←GP(x,s)
(3) MATCH←M(y,S)
(5) MATCH←M(x,R)
(6) MATCH←M(x,J)
M(R,S)←
M(R,M)←
M(B,F)←
(3) MATCH←F(y,S)
(5) MATCH←F(x,R)
(6) MATCH←F(x,J)
F(H,J)←
F(H,G)←
F(J,S)←
Extensional Database

Fig. 9

Control of Logic Programs Using Integrity Constraints

Madhur Kohli Jack Minker

Department of Computer Science
University of Maryland
College Park, MD 20742

Abstract

This paper presents a theory for the intelligent execution of function free logic programs.

Generally, interpreters for logic programs have employed a simple search strategy for the execution of logic programs. This control strategy is 'blind' in the sense that when a failure occurs, no analysis is performed to determine the cause of the failure and to determine the alternatives which may avoid the same cause of failure.

When executing a logic program it is often desirable to permit an arbitrary selection function and to have several active nodes at any given time. It is also useful to be able to remember the causes of failures and use this information to guide the search process.

In this paper we present a theory for using integrity constraints, whether user supplied or automatically generated during the search, to improve the execution of function free logic programs. Integrity constraints are used to guide both the forward and backward execution of the programs. The theory supports arbitrary node and literal selection functions and is thus transparent to the fact whether the logic program is executed sequentially or in parallel.

1. Introduction

1.1. The Problem

This paper presents a theory for the intelligent execution of function free logic programs.

Interpreters for logic programs have employed, in the main, a simple search strategy for the execution of logic programs. PROLOG (Roussel [1975], Warren [1979], Roberts [1977]), the best known and most widely used interpreter for logic programs, employs a straightforward depth first search strategy augmented by chronological backtracking to execute logic programs. This control strategy is 'blind' in the sense that when a failure occurs, no analysis is performed to determine the cause of the failure and to determine the alternatives which may avoid the same cause of failure. Instead the most recent node where an alternative exists, is selected. This strategy has the advantage that it is efficient in that no decisions need to be made as to what to select next and as to where to backtrack. However, the strategy is extremely inefficient when it backtracks blindly and thus repeats failures without analyzing their causes.

Pereira [1982], Bruynooghe [1978] and others have attempted to improve this situation by incorporating the idea of intelligent backtracking within the framework of the PROLOG search strategy. In their work the forward execution component remains unchanged, however, upon failure their systems analyze the failure and determine the most recent node which generated a binding which caused the failure. This then becomes the backtrack node. This is an improvement over the PROLOG strategy but still suffers from several drawbacks. Their scheme works only for a depth first search strategy and always backtracks to the most recent failure causing node. Also, once the backtrack node has been selected, all information about the cause of the failure is discarded. This can lead to the same failure in another branch of the search tree. Pietrzykowski [1981] has considered intelligent backtracking in the framework of general first-order systems.

A node in the search space is said to be closed when it has provided all the results possible from it. In most PROLOG based systems a node cannot be closed until every alternative for that node is considered. However, by using integrity constraints as will be shown later, a node can be closed once it is determined that exploring further alternatives for that node will not provide any more results.

When executing a logic program it is often desirable to permit an arbitrary selection function and to have several active nodes at any given time. It is also useful to be able to remember the causes of failures and use this information to guide the search process.

In this paper we present a theory for using integrity constraints, whether user supplied or automatically generated during the search, to improve the execution of function free logic programs. Integrity constraints are used to guide both the forward and backward execution of programs. The theory supports arbitrary node and literal selection functions and is thus transparent as to whether the logic program is executed sequentially or in parallel.

In the rest of this section we define the class of logic programs to which this theory is applicable. In Section 2 we show how integrity constraints can be used to guide the forward execution of the system. In Section 3 we show how integrity constraints can be extracted from failure and propagated up the search tree. In the Appendix we present the interpreter for this theory.

1.2. Function Free Logic Programs

1.2.1. Horn Clauses

Horn clauses are a subset of the first order predicate calculus. The language of function free Horn clauses is defined below.

A term is a constant or variable.

An atomic formula is a predicate letter of arity $n \geq 1$ whose arguments are terms, i.e., if P is an n-ary predicate letter and $t_1,\ldots,t_n$ are terms then $P(t_1,\ldots,t_n)$ is an atomic formula.

An atomic formula or its negation is a literal. The classical logical connectors ~ (not), /\ (and), V (or) and the universal quantifier ∀ are used in constructing clauses.

A clause is a disjunction of literals all of whose variables are universally quantified. That is, $B_1 V \ldots V B_m V \sim A_1 V \ldots V \sim A_n$ is a clause. A clause can be written equivalently as $(\forall x_i)\ (B_1 V \ldots V B_m \leftarrow A_1 /\backslash \ldots /\backslash A_n)$ where the x_i, i=1,...,k are all the variables in the atomic formulae A_i, B_j, i=1,...,n, j=1,...,m. Since all variables in a clause are universally quantified, the universal quantifier will be omitted in the rest of this paper.

A Horn clause is a clause which has at most one positive literal, i.e., the clause above is Horn iff $m \leq 1$.

A function free logic program is composed of function free Horn clauses as follows:

1. Assertions are facts or general statements about the domain and are of the form
$$P(x_1,\ldots,x_m) \leftarrow$$
2. Procedures are of the form:
$$P(x_1,\ldots,x_m) \leftarrow P_1(\ldots),\ldots,P_n(\ldots)$$
which states that to solve P we must solve $P_1,\ldots,P_n$.
3. Goals or the problem to be solved are of the form
$$\leftarrow P(\ldots),Q(\ldots),\ldots,S(\ldots)$$

A function free logic program is defined in terms of the following:

(1) Axioms

(a) Domain Closure Axioms, which states that there is a finite set of constants $c_1,\ldots,c_n$ from which all constants in the knowledge base must be drawn.

(b) Unique Name Axioms, which state that the constants are unique i.e., $(c_1 \neq c_2)$, ..., $(c_1 \neq c_n)$, ..., $(c_{n-1} \neq c_n)$

(c) Equality Axioms:

reflexive $(x = x)$

symmetric $((y = x) \leftarrow (x = y))$

transitive $((x = z) \leftarrow (x = y) \wedge (y = z))$

principle of substitution of equal terms

$$P(y_1,\ldots,y_n) \leftarrow P(x_1,\ldots,x_n) \wedge (x_1=y_1) \wedge \ldots \wedge (x_n = y_n)$$

where P is an n-ary predicate letter.

(2) Assertions of the form

$$P(x_1,\ldots,x_n) \leftarrow$$

(3) Procedures of the form

$$P(x_1,\ldots,x_n) \leftarrow P_1(\ldots),\ldots,P_m(\ldots)$$

(4) A meta-rule: Negation as failure to prove positive literals. ([Clark 1978] and [Reiter 1978]).

1.2.2. Integrity Constraints

An integrity constraint is an invariant that must be satisfied by the clauses in the knowledge base. That is, if T represents a theory of function free logic programs and IC represents a set of integrity constraints applicable to T, then T U IC must be consistent.

Integrity constraints are closed function free Horn formulae of the form:

(a) $\leftarrow P_1,\ldots,P_m$, or

(b) $Q \leftarrow P_1,\ldots,P_m$, or

(c) $E_1 \vee E_2 \vee \ldots \vee E_n \leftarrow P_1,\ldots,P_m$ where the E_i, i=1,...,n, are equality predicates i.e. each E_i is of the form $x_i = y_i$ where at least one of the x_i, y_i are variables.

Thus, an integrity constraint of the form (a) above, represents negated data, in the sense that $P_1 \wedge P_2 \wedge \ldots \wedge P_m$ can never hold if T U IC is consistent.

An integrity constraint of the form (b) above, states that if

$$P_1 \wedge P_2 \wedge \ldots \wedge P_m$$

holds then Q must also hold.

Integrity constraints of the form (c) above, represent dependencies between the arguments of $P_1,P_2,\ldots,P_m$.
Consider the logic program:

$$\left.\begin{array}{l} P(x,y)\leftarrow F(x,y) \\ F(a,b)\leftarrow \\ F(b,c)\leftarrow \end{array}\right| T$$

and the associated integrity constraint

$$R(x,z) \leftarrow P(x,y), P(y,z) \mid IC$$

In the above example, $\overline{R(a,c)}$ can be proven from T, by using negation by failure. P(a,b) and P(b,c) can be proven from T. R(a,c) can be proven from P(a,b) and P(b,c) and IC. Thus R(a,c) can be proven from T U IC. Thus T U IC is inconsistent. The above integrity constraint is violated by the logic program since T U IC is inconsistent.

If, however, the clause:

$$R(x,z) \leftarrow F(x,y), F(y,z)$$

were added to T, then T U IC would be consistent and the integrity constraint would not be violated. Similarly, one could leave the axiom as an integrity constraint and add R(a,c) to the knowledge base and have a consistent theory.

2. Goals and Integrity Constraints

2.1. Integrity Constraints to Limit Forward Execution

Though integrity constraints are not necessary for finding the solution of a given set of goals with respect to a given logic program (Reiter [1978]), they can greatly enhance the efficiency of the search process and thus improve the performance of the problem solver (McSkimin and Minker [1979], King [1981]).

Integrity constraints enable the semantics of the given domain to direct the search strategy by enabling the problem solver to prune those alternatives which violate integrity constraints and thus focus the search. Thus, integrity constraints influence the forward execution of the problem solver by enabling it to detect which sets of goals are unsolvable. This avoids exploring alternatives which must fail after a, possibly lengthy, full search.

Thus whenever a new set of subgoals is generated, this set can be tested to determine if it violates any integrity constraints. If so, the node in question can be discarded and another path considered.

2.2. Implementation and Search Strategy

There are several forms an integrity constraint may take (Section 1.2.2).

Whenever a new set of goals is generated it must be tested to determine if it violates an integrity constraint. Though each of the forms (a), (b), and (c) above require slightly different treatments to determine if they are violated, the underlying mechanism for each is the same.

Form (c) can be transformed into form (a) by moving the disjunction of equalities on the left into a conjunction of inequalities on the right, i.e.,

$$E_1 \vee E_2 \vee \ldots \vee E_n \leftarrow P_1, \ldots, P_m$$

is equivalent to

$$\leftarrow P_1, \ldots, P_m, \overline{E_1}, \overline{E_2}, \ldots, \overline{E_n}.$$

These inequalities can then be handled by using predicate evaluation rather than negation.

Form (b) can be interpreted to mean that solving Q is equivalent to solving $P_1,\ldots,P_m$ and thus $P_1,\ldots,P_m$ can be replaced by Q in the set of goals.

Since all that is required, is to determine if the literals in the integrity constraint occur in the goal clause, an extremely straightforward algorithm can be used. It is only necessary to determine if the right hand side of some integrity constraint can subsume the goal clause.

A clause C subsumes a clause D iff there exists a substitution σ such that

$$C\sigma \subseteq D$$

By $C\sigma$ we mean the result of applying a substitution set σ, which is composed of pairs of the form a_i/x_i where the x_i are variables of C and the a_i are variables or constants, to C, i.e. each occurrence of x_i in C is replaced by a_i.

The subsumption algorithm executes in linear time and does not increase the complexity of the search.

This algorithm (Chang and Lee [1973]) is presented below:

Let C and D be clauses.
Let $\theta = \{a_1/x_1,\ldots,a_n/x_n\}$ be a substitution set, where $x_1,\ldots,x_n$ are all the variables occurring in D and $a_1,\ldots,a_n$ are new distinct constants not occurring in C or D (Skolem constants) and the constants a_i is to be substituted for the variable x_i, wherever it appears in C.
Suppose $D = L_1 \vee L_2 \vee \ldots \vee L_m$,
then $D\theta = L_1\theta \vee L_2\theta \vee \ldots \vee L_m\theta$
$D\theta$ is a ground clause since all variables in D have been replaced by Skolem constants.
Thus, $\sim D\theta = \sim L_1\theta \wedge \sim L_2\theta \wedge \ldots \wedge \sim L_m\theta$

1: Let $W = \{\sim L_1\theta,\ldots,\sim L_m\theta\}$

2: Let $k = 0$ and $U^0 = \{C\}$

3: If U^k contains the null clause then terminate; C subsumes D
else let U^{k+1} = {resolvents of C_1 and C_2 | $C_1 \in U^k$ and $C_2 \in W$}

4: If U^{k+1} is empty then terminate; C does not subsume D
else set k to k+1; go to Step 3.

Consider now, how the various forms of integrity constraints can be used to limit the forward execution.

If a constraint is a form (a) constraint then all that is required is to apply the subsumption algorithm to the newly generated goal clause. If the constraint subsumes the goal clause, the goal violates the constraint and should be deleted from the search space.

Whenever a literal is solved, it must be determined whether it unifies with any literal in the right hand side of a form (c) constraint. If so, the

resulting substitution is applied to the constraint, the solved literal is deleted, and the resulting clause is added to the set of integrity constraints.
For example, if

$$x_1=x_2 \leftarrow P(x,x_1),P(x,x_2)$$

is a constraint and P(a,b) is solved then P(a,b) unifies with the right hand side of the above constraint with the substitution set $\{a/x, b/x_1\}$. Applying this substitution to the above constraint, and noticing that P(a,b) has been solved permits the revised constraint

$$x_2=b \leftarrow P(a,x_2)$$

to be obtained. This is then added to the set of integrity constraints. Also, this allows any node containing P(a,x) to be considered as a purely deterministic node, since only one possible solution for P(a,x) exists.

Finally form (b) constraints can be used as follows. If the right hand side of the constraint subsumes the goal then, the resulting substitution is applied to the left hand side of the constraint and a new alternative goal with the left hand side substituted for the right hand side of the constraint, is generated. For example, if

$$Q(x,z) \leftarrow P_1(x,y),P_2(y,z)$$

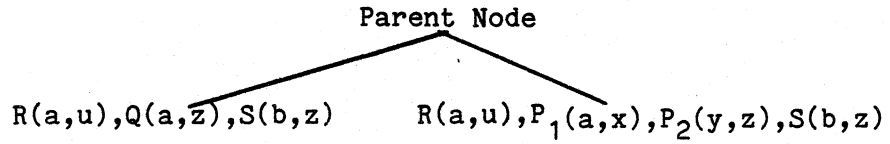
is a constraint, and the goal clause under consideration is

$$\leftarrow R(a,u),P_1(a,y),P_2(y,z),S(b,z)$$

then

$$\leftarrow P_1(x,y),P_2(y,z)$$

subsumes the goal with substitution $\{a/x\}$. Applying this substitution to $\leftarrow Q(x,z)$ results in $\leftarrow Q(a,z)$. Generating an alternative node with Q replacing P_1,P_2 then results in the above goal node being replaced by the following OR-node



Whenever a violation of an integrity constraint occurs it is treated as a failure. This results in failure analysis and backtracking which are detailed in the next section.

3. Local and Global Conditions

Global conditions are integrity constraints which are applicable to every possible node in the search space. Local conditions are integrity constraints which are generated during the proof process and which are applicable only to the descendant nodes of some given node in the search space.

In this section we show that both local and global conditions exist. We also show how implicit global and local conditions may be determined and how they can be used to improve the efficiency of a problem solver. We also show how both failure nodes and fully expanded nodes may be used to derive these conditions.

3.1. Failure

The failure of a literal can provide valuable information for directing the search. A literal 'fails' when it cannot be unified with the head of any clause (intensional or extensional) in the knowledge base. Since this failure means that the literal cannot be proven in the current knowledge base, because of the assumption of failure by negation, the literal's negation can be assumed to hold. Thus, the negation of the literal can be viewed as an implicit integrity constraint, and the failure can be viewed as a violation of this integrity constraint.

Thus, every failure can be viewed as a violation of some integrity constraint, implicit or explicit. This allows us to extract useful information from every failure, and to use this information in directing the search.

The possible causes of unification conflicts are:

(a) The literal is a pure literal. That is, there is no clause in the knowledge base, which has as its head the same predicate letter as the literal selected. This implies that any literal having the same predicate letter as the selected literal, will fail anywhere in the search space. This information can be useful in terminating other branches of the search tree in which a literal containing this predicate letter occurs. Thus if P(a,x) is a pure literal, then all of its argument positions can be replaced by distinct variables and the resulting literal can be added to the set of integrity constraints as a form (a) constraint, i.e.,

$$\leftarrow P(x_1,x_2)$$

is added to the set of constraints.

(b) There are clauses in the knowledge base which could unify with the selected literal, but which do not unify because of a mismatch between at least two constant names. This mismatch can occur in two ways:

(i) one of the constant names occurs in the literal and the other occurs textually in the head of the clause with which it is being matched.

(ii) both the constants, which are distinct, occur in only one of the literal or the clause head being matched, but are to be bound together by the repeated occurrence of a variable in the clause head or literal, respectively.

In both cases (i) and (ii) it is obvious that the selected literal can never succeed with that particular set of arguments. This information can be used as an integrity constraint by placing the literal as a form (a) constraint. For example, if the selected literal is

$$P(x,a,x)$$

and the only P clauses in the knowledge base are

$$P(nil,nil,nil) \leftarrow$$
$$P(z,z,b) \leftarrow P_1(z,b),P_2(z)$$

then the unification fails and $\leftarrow P(x,a,x)$ can be added to the set of integrity constraints.

3.2. Explicit and Implicit Integrity Constraints

Integrity constraints may be either explicit or implicit. Explicit integrity constraints are those which are provided initially in the domain specification. These constraints affect the forward execution of the problem solver as detailed in Section 2. These constraints can also be used in the derivation of implicit constraints.

Implicit integrity constraints are generated during the proof process, i.e., during the solution of a specific set of goals. These constraints arise out of the information gleaned from failure as shown in section 3.1, and from successes in certain contexts as will be shown in later sections. These integrity constraints may be considered to be implicit integrity constraints in the sense that they are not explicitly supplied integrity constraints but are derived from the proof process.

3.3. Applicability of Integrity Constraints

An integrity constraint may be globally or locally applicable. An integrity constraint is said to be globally applicable if it can be applied to any node in the search space. That is, it must be satisfied by every node on every success path in the proof tree. Explicit integrity constraints are always globally applicable since they are defined for the domain and are independent of any particular proof tree. Implicit integrity constraints may be either locally or globally applicable.

A locally applicable integrity constraint is one which must be satisfied by a given node and all its children. Any node which is not part of the subtree rooted at the node to which the constraint is locally applicable, need not satisfy the constraint. Locally applicable integrity constraints are derived from the failure of some path in the search space. The analysis of the cause of the failure results in the generation of a locally applicable integrity constraint which is transmitted to the parent node of the failure node. This local integrity constraint must then be satisfied by any alternative expansions of the node to which it applies. This effectively prunes those alternatives which cannot satisfy the constraint. For example, consider the following logic program fragment,

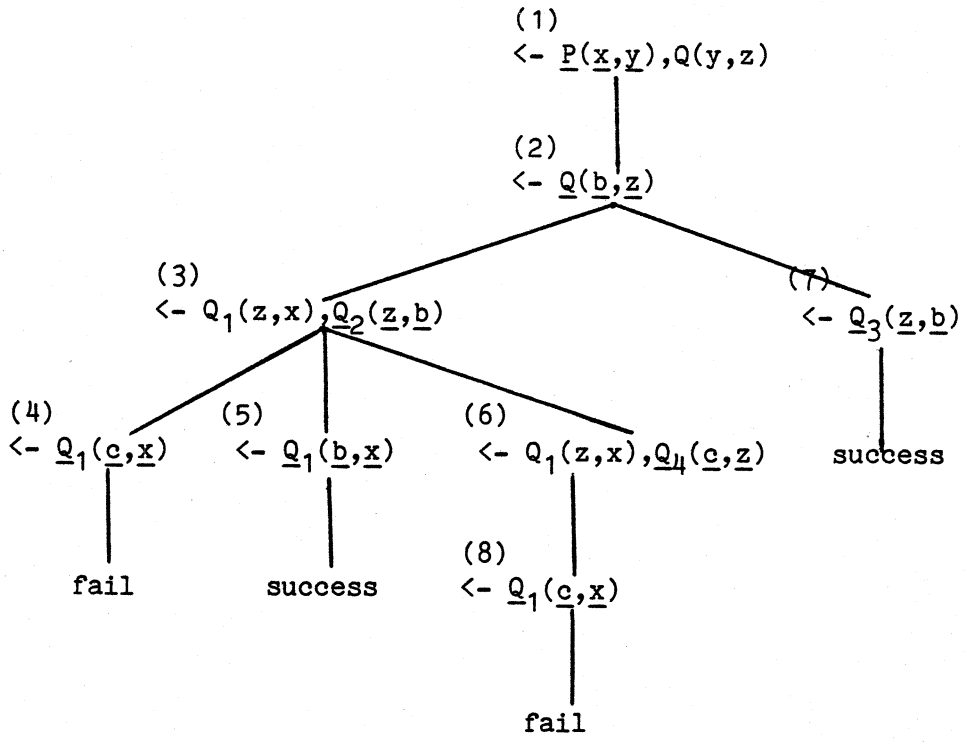
Logic Program:

$P(a,b)$ <-
$Q(y,z)$ <- $Q_1(z,x)$, $Q_2(z,y)$
$Q(y,z)$ <- $Q_3(z,y)$
$Q_1(b,d)$ <-
$Q_2(b,b)$ <-
$Q_2(c,c)$ <-
$Q_2(c,b)$ <-
$Q_2(x,y)$ <- $Q_4(c,x)$
$Q_3(c,b)$ <-
$Q_4(x,x)$ <-

Query:

<- $P(x,y),Q(y,z)$

Search Tree:



From the proof tree, the following information can be exteracted. From node 4, <- $Q_1(c,x)$ can be propagated as a global implicit constraint since <- $Q_1(c,x)$ can never be solved. Also, z = c can be propagated as a local implicit constraint to node 3 and thus later prevent the generation of node 8. This constraint is local to node 3 and its children since that is the node that bound z to c. Thus, node 6 has inherited this local constraint and thereby prevents z from being bound to c. As can be seen from the example an alternative expansion of node 2 giving node 7 succeeds with z bound to c, which illustrates that z = c at node 3 is a local constraint.

3.4. Fully Expanded Nodes

Implicit integrity constraints can be derived not only from failures but also from fully expanded nodes.

If the same local integrity constraint is generated by every expansion of a given node, then this constraint can be propagated globally, irrespective of whether or not some expansion of this node succeeded. Also any node which fails for every possible expansion, can be propagated as a global integrity constraint.

3.5. Generation and Propagation of Conditions

Implicit integrity constraints are generated at the leaf nodes of the search space and are then propagated either globally or as locally applicable integrity constraints to some parent node of the leaf node. The rules for generating and propagating these dynamic constraints are detailed below.

When a goal fails along all paths, then that goal along with its current bindings is propagated as a global integrity constraint. Thus, if $P(\alpha_1, \alpha_2 \ldots, \alpha_n)$, where the α_i, $i = 1,\ldots,n$ are constants or variables, fails for every expansion of P, then $\leftarrow P(\alpha_1, \alpha_2, \ldots, \alpha_n)$ is a global integrity constraint. This is because $P(\alpha_1, \alpha_2, \ldots, \alpha_n)$ can never succeed, given the current state of the knowledge base.

Since that goal can never succeed with its current bindings, alternatives which give rise to different bindings for its arguments must be tried. Thus those nodes which created the failure causing bindings receive as local integrity constraints, the information that these bindings must not be repeated along alternative expansions of the nodes which created the bindings. That is, if $P(\alpha_1, \alpha_2, \ldots, \alpha_n)$ fails and there is some ancestor P' of P such that some α_i of P is bound by some literal (other than P') in the clause containing P', then $\leftarrow x_i = \alpha_i$ is a local integrity constraint for the clause containing P'. If there are several α_i which have been bound in different clauses then the conjunction of these bindings must be propagated to the binding clauses. That is, if $\beta_1, \beta_2 \ldots, \beta_m$ are constants, where $\beta_i = \alpha_j$ $i = 1,\ldots,m$; $j = 1,\ldots,n$, such that β_i was bound by some ancestor P_i of P, and if P_1 is the most recent ancestor of P and P_m is the least recent ancestor of P, then $\leftarrow x_1 = \beta_1$ is propagated to the clause containing P_1 and $\leftarrow x_1 = \beta_1, x_2 = \beta_2, \ldots, x_i = \beta_i$ is propagated to the clause containing P_i. This is because undoing the β_1 binding at P_1 may suffice to remove the cause of failure whereas at P_2, undoing either of the β_1 or β_2 bindings may suffice. We do not propagate $\leftarrow x_1 = \beta_1, x_2 = \beta_2, \ldots, x_i = \beta_i$ to every P_k since x_i has been replaced by β_i in P_i and thus does not occur in any P_k, $k < i$.

Local constraints which are propagated to a node by a descendant of the node must then be propagated to all other descendants of that node. This is because, as was noted above, the binding of β_i to x_i in the node containing P_i was due to the selection of some atom other than P_i in that node. Thus, P_i will be present in every expansion of that node and the binding of β_i to x_i will cause P_i to eventually fail.

Theorem:

Consider a node P which has several children $P_1, P_2, \ldots, P_n$.
Associated with each P_i is a set of local integrity constraints generated by its descendent nodes.
Let IC_i be the set of local integrity constraints associated with each P_i. Then $\bigcap_i IC_i$ is propagated to P.

Proof :

Let IC_i be the set of local integrity constraints applicable to P_i, i.e.,

IC_i is the set of constraints generated by the descendents of P_i using the rules explained above.

Let IC_P be the set of local integrity constraints applicable to P, then $\bigcap_i IC_i \subseteq IC_P$. Let $\gamma \lessdot \bigcap_i IC_i$, then $\gamma \lessdot IC_i$ $\forall$ i=1,...,n and thus every P_i will fail for any binding that satisfies $\bar{\gamma}$. Thus since every P_i fails, P must fail with any binding that satisfies $\bar{\gamma}$. Thus $\gamma \lessdot IC_P$ i.e. $\gamma \lessdot \bigcap_i IC_i \Rightarrow \gamma \lessdot IC_P$ and $\bigcap_i IC_i \subseteq IC_P$.

4. Summary

A theory has been developed for function-free logic programs to permit control of the search based both on domain specific information in the form of integrity constraints and on an analysis of failures. Integrity constraints limit search in the forward direction, while failures result in the creation of integrity constraints. Failure analysis is also used to determine backtrack points which are more likely to succeed. The concepts of local and global constraints have been introduced and are to be used to inhibit exploring fruitless alternatives. Subsumption is employed to take advantage of the constraints. A logic program is provided for an interpreter which will perform the above.

We intend to incorporate these concepts into PRISM, a parallel logic programming system [Kasif,Kohli and Minker 1983], under development at the University of Maryland.

5. Acknowledgements

This work was supported in part by AFOSR grant 82-0303 and NSF grant MCS-79-19418.

6. References

[Bruynooghe 1978]
Bruynooghe, M., Intelligent Backtracking for an Interpreter of Horn Clause Logic Programs, Report CW 16, Applied Math and Programming Division, Katholieke Universiteit, Leuven, Belguim, 1978.

[Chang and Lee 1973]
Chang, C.L., and Lee, R.C.T., Symbolic Logic and Mechanical Theorem Proving, Academic Press, New York, 1973.

[Clark 1978]
Clark, K.L., "Negation as Failure", in Logic and Databases, H. Gallaire and J. Minker, Eds., Plenum Press, New York, 1978, pp 293-322.

[Kasif, Kohli, and Minker 1983]
Kasif, S., Kohli, M., and Minker, J., PRISM: A Parallel Inference System for Problem Solving, Technical Report, TR-1243, Dept. of Computer Science, University of Maryland, College Park, 1983.

[King 1981]
King, J.J., Query Optimization by Semantic Reasoning, Ph.D Thesis, Dept of Computer Science, Stanford University, May 1981.

[McSkimin and Minker 1977]
McSkimin, J.R., and Minker, J., The Use of a Semantic Network in a Deductive Question Answering System, Proceedings IJCAI-77, Cambridge, MA, 1977, pp 50-58.

[Pereira 1982]
Pereira, L.M., and Porto, A., Selective Backtracking, in Logic Programming, K.L. Clark and S-A. Tarnlund, Eds., Academic Press, New York, 1982, pp 107-114.

[Pietrzykowski 1982]
Pietrzykowski, T., and Matwin, A., Exponential Improvement of Efficient Backtracking: A Strategy for Plan Based Desduction, Proceedings of the 6th Conference on Automated Deduction, Springer Verlag, New York, June 1982, pp 223-239.

[Reiter 1978]
Reiter, R., On Closed World Data Bases, in Logic and Databases, H. Gallaire and J. Minker, Eds., Plenum Press, New York, 1978, pp 55-76.

[Roberts 1977]
Roberts, G.M., An Implementation of PROLOG, M.S. Thesis, University of Waterloo, 1977.

[Roussel 1975]
Roussell, P., PROLOG: Manuel de Reference et d'Utilisation. Groupe d'Intelligence Artificielle, Universite d'Aix-Marseille, Luminy, 1975.

[Warren 1979]
Warren, D.H.D., Implementing PROLOG: Compiling Predicate Logic Program, Department of Artificial Intelligence, University of Edinburgh. Research Reports 39 and 40, 1979.

Appendix: The Interpreter

In this appendix we describe the interpreter, for function-free logic programs, which implements the theory described in the previous sections.

Defn: An open node is a node which has not been selected for expansion.

Defn: An active node is a node which has been selected for expansion but has not yet been fully solved.

Defn: A closed node is a node which has been completely expanded.

1. Interpreter Specification

1. Initialise the search space with the initial goals.
2. Select an open node from the search space. Mark it as active.
3. If the selected node is subsumed by a global integrity constraint then go to step 15.
4. Select an atom from the selected node.
5. Unify the selected atom with all procedure heads which have the same predicate letter.
6. Delete all procedures which require bindings which violate the local integrity constraints of the selected node, from the set of procedures which were selected in step 5.
7. If the set of procedures obtained after step 6 is empty, go to step 14.
8. Add the set of selected procedures to the set of all selected but unconsumed procedures generated so far.
9. Select one or more procedures from the set of unconsumed procedures.
10. For each of the procedures selected in step 8: generate descendant nodes of the node which was expanded to obtain these procedures. The descendants are generated by replacing the selected literal by the body of a matching procedure, and applying the substitution obtained from the unification process to the newly generated node. Mark the newly generated node as open.
11. If any of the newly generated nodes is empty (null clause), STOP.
12. Initialise the local constraint sets for the newly generated nodes, with the local constraint sets of their parent node.
13. go to step 2.
14. Add the selected node with all its current bindings to the list of global integrity constraints.
15. Mark the current node as closed. For each ancestor node which made a binding in the current node, add a local constraint which is the negation of the bindings.
16. Propagate the local constrint set to all children of each affected node.
17. Go to step 2.

2. Logic Program for the Interpreter

In this section we define the logic program for the interpreter described in the previous section.

A node is a 3-tuple

n(state,clause,localic)

where, state is either open, active or closed; clause is the set of goals represented by this node; localic is the set of local integrity constraints.

A body is a 2-tuple,

b(clause,sublist)

where, clause is the procedure body; and, sublist is the substitution set generated by unification of the literal and the procedure head.

```
solve(goals,prog,ic) <-
          init(goals,tree),
          interpret(tree,prog,ic).

init(goals,tree) <-
          inittree(n(open,goals,nil),tree).

interpret(tree,prog,ic) <-
          empty(tree).
interpret(tree,prog,ic) <-
          selectnode(tree,node,newtree1),
          expandnode(node,newtree1,prog,ic,newtree,newic),
          interpret(newtree,prog,newic).

expandnode(node,tree,prog,ic,newtree,newic) <-
          icok(node,ic),
          selectatom(node,atom),
          unify(atom,prog,bodies),
          checklocalic(node,bodies,newbodies),
          insertbodies(newbodies,node,atom,tree,ic,newtree,newic).

expandnode(node,tree,prog,ic,newtree,ic) <-
          not(icok(node,ic)),
          failurenode(node,tree,newtree).

icok(node,ic) <-
          empty(ic).
icok(node,ic) <-
          selectic(ic,oneic,restic),
          not(subsume(oneic,node)),
          icok(node,restic).

checklocalic(node,nil,nil,ic,ic) <- .
checklocalic(n(x,y,lic),b(z,subs).restbodies,b(z,subs).newbodies,ic,newic) <-
          checklicok(lic,subs),
          checklocalic(n(x,y,lic),restbodies,newbodies,ic,newic).
```

failurebindings(node,bindings) creates a list of bindings in node, undoing any one or more of which may undo the cause of failure.

addlocalic(parents,bindings,newparents) updates the local integrity constraint sets for parents with those elements of bindings which are applicable, and creates the new node list newparents.

propagatelocalic(parents,tree,newtree) propagates the local integrity constraint sets of each element of parents to every child of that node. It generates a newtree with all the newly modified nodes.

addic(node,ic,newic) adds node to the set of global integrity constraints ic to generate the new global integrity constraint set newic.

selectic(ic,singleic,restic) selects an integrity constraint singleic from the set of integrity constraints ic and creates the set restic which is ic - singleic.

```
checklocalic(n(x,y,lic),b(z,subs).restbodies,newbodies,ic,newic) <-
          not(checklicok(lic,subs)),
          addic(z,ic,newic1),
          checklocalic(n(x,y,lic),restbodies,newbodies,newic1,newic).

checklicok(nil,subs) <-
checklicok(lic.restlic,subs) <-
          not(member(lic,subs)),
          checklicok(restlic,subs).

insertbodies(bodies,node,literal,tree,ic,newtree,ic) <-
          not(empty(bodies)),
          inserttree(node,literal,bodies,tree,newtree).
insertbodies(bodies,node,literal,tree,ic,newtree,newic) <-
          empty(bodies),
          addic(node,ic,newic),
          failurenode(node,tree,newtree).

failurenode(node,tree,newtree) <-
          bindingnodes(node,tree,parents),
          failurebindings(node,bindings),
          addlocalic(parents,bindings,newparents),
          updatetree(parents,newparents,tree,newtree1),
          propagatelocalic(newparents,newtree1,newtree).
```

where,

inittree(node,tree) creates tree with node as its root node.

inserttree(node,literal,bodies,tree,newtree) creates a newtree which is formed from tree by generating children nodes of node by replacing literal in node by the atoms in each of the bodies and applying the substitutions to the newly generated nodes.

updatetree(nodes,newnodes,tree,newtree) modifies tree to produce newtree by replacing each element of nodes in the tree by the corresponding element in newnodes.

selectnode(tree,node,newtree) selects an open node from tree and creates a newtree which has node replaced by a new node marked as active.

selectatom(node,atom) selects an atom from node.

unify(atom,prog,bodies) selects those procedures from prog which have the same name as atom. It then applies the unification algorithm to each of these procedure heads and forms a list of those bodies which match the given atom, along with the resulting substitution lists.

subsume(ic,node) determines whether the integrity constraint ic subsumes node.

member(element,set) determines if element is a member of set.

empty(list) determines if list is empty(nil).

bindingnodes(node,tree,parents) sets parents to be the list of all ancestors of node which created a binding in node.

171

Interprocess Communication in Concurrent Prolog

Akikazu Takeuchi Kouichi Furukawa
Research Center
Institute for New Generation Computer Technology
Mita-Kokusai Building, 21F.
4-28, Mita 1-chome, Minato-ku, Tokyo 108
Japan

Abstract

Concurrent Prolog is a logic-based concurrent programming language which was designed and implemented on DEC-10 Prolog by E. Shapiro. In this paper, we show that the parallel computation in Concurrent Prolog is expressed in terms of message passings among distributed activities and that the language can describe parallel phenomena in the same way as Actor-formalism does. Then we examine the expressive power of communication mechanism based on shared logical variables and show that the language can express both unbounded buffer and bounded buffer stream communication only by read-only annotation and shared logical variables. Finally the new feature of Concurrent Prolog is presented, which will be very useful in describing the dynamic formation and reformation of communication network.

1. Introduction

Concurrent Prolog was designed and implemented on the DEC-10 Prolog by E. Shapiro [1] for concurrent programming. As the Relational Language [2], Concurrent Prolog adopts Or-parallelism as a basis for non-deterministic processing, and And-parallelism for description for parallel processes. Shared variables are used, with some control information "variable annotation", as communication channels among concurrent processes.

In the Relational Language, there are two kinds of the variable annotation, input and output, which are used for input suspension and output suspension respectively. On the contrary, in Concurrent Prolog, there is only one annotation, read-only annotation, which is a generalized idea of input annotation and by which we can also express the output suspension when an output buffer is full. This will be explained in the section 4.

In the section 2, we review the Concurrent Prolog. In the section 3, the computation model of the language is presented and in the section 4 we examine the basic communication mechanism based on shared logical variables and derive the technique for implementing the bounded buffer communication in the language. In the section 5, we introduce the concept of the incomplete message as a new programming paradigm and explain briefly. In the section 6, we present a new feature of Concurrent Prolog which is very useful in describing the formation and reformation of the communication network.

2. Review of the Concurrent Prolog

2.1 Syntax of Concurrent Prolog

In Concurrent Prolog, a program is represented as a list of guarded clauses. The form of a guarded clause is

A :- G1,...,Gn | B1,...,Bm. n,m >= 0.

A guarded clause must have a guard bar "|". The left hand side of guard bar is called the guard sequence and the right hand side is called the goal sequence. The guard bar can be omitted when the guard sequence is empty, that is n=0. G's and B's are both lists of literals connected by logical AND.

There are two kinds of logical AND's, which are parallel-AND and serial-AND.

serial-AND	"&"
parallel-AND	","

Their logical meaning are the same, but the way to interpret and execute is different. As it is clear from their name, goals connected by serial-AND must be executed in sequential order (left-to-right), and goals connected by parallel-AND must be executed in parallel. As for the operator precedence,

"," is lower than "&", that is,

f&g , p&q is equivalent to (f&g) , (p&q).

Current implementation of Concurrent Prolog only provides sequential-or mode. Therefore, alternative clauses are tried in the text order.

On the notation, we adopt DEC-10 Prolog-like convention, for example, a word beginning with a capital letter denotes a variable.

In Concurrent Prolog, variables can be accompanied with some special control information, "read-only" annotation, which can control the unification. Read-only annotation is denoted by "?" and can be attached to variables in the following way,

X? where X is a variable.

The meaning of read-only annotation is that a variable annotated by "?" must not be unified with a non-variable term. The annotation can be attached to each occurence of a variable, and will vanish when the variable will be instantiated to a non-variable term. Generally read-only annotation can be attached to the variables shared by concurrent processes in order to restrict the direction of data flow, where the process which annotates the shared variable can not instantiate the variable and wait for the variable to become instantiated by the other process which does not annotate it. This will be explained later again.

2.2 Reduction

In this section, the process of reduction is explained. Suppose that the goal A and the following program are given.

```
A1 :- G1 | B1.
A2 :- G2 | B2.
      .
      .
      .
An :- Gn | Bn.
```

where Gi and Bi (1 =< i =< n) are a guard sequence and a goal sequence respectively.

Each clause is classified into one of the three following classes with respect to the goal A.

1. Candidate Ai :- Gi | Bi.

   when, without instantiating variables annotated by "?" to non-variable terms, A and Ai can be unified and Gi can be solved.

2. Suspended Aj :- Gj | Bj.

when, except for instantiating read-only variables to non-variable terms, A and Aj can be unified and Gj can be solved.

3. Failure
otherwise.

Each clause is checked in a text order whether it can be a candidate, and the clause that is found to be a candidate first is selected. The selected clause, say A :- GilBi., is used to reduce the goal to the goal sequence Bi. Once the goal is reduced, checking of the rest of clauses will be abandoned. In this sense, the guard bar "|" acts as a cut symbol.

When the goal has no candidate and has at least one suspended clause, it will be suspended until at least one candidate will be found or it will be failed (i.e. all the clauses will be classified into the failure).

Since the instantiation of shared variables can be undone by backtracking before the guard sequence is solved completely, the values of the shared variables will be hidden from other processes until the guard sequence is solved completely.

Although Concurrent Prolog adopts And-parallelism, consistency check of values of shared variables will be replaced by the restriction that the process instantiating the shared variables must be one. However, which process can instantiate a shared variable need not be specified before the execution, as long as it is guaranteed that there can be only one such process even if it is determined dynamically in a non-deterministic way.

3. The Computation Model

In this section we present the Actor-like model [3,4] of the parallel computation in Concurrent Prolog. For the simplicity, we assume that every goals are solved in Or-parallel mode, that is, all the alternatives are checked in parallel.

First we define the term "event" which is a basic concept in order to formalize the computation model.

"An event is a successful unification between a goal and a head of a clause and a successful solution of the guard sequence of that clause."

Using this definition, we can specify the condition for an event to arise.

"The condition for an event associated with a goal to arise is that the goal can be unified with a head of some clause and its guard sequence can be solved successfully."

Given a goal A and a clause A' :- G1,...,Gn|B1,...,Bm, we denote the event by

A:A'.

Once a goal A is unified with the head A' of a clause

A' :- G1,...,Gn|B1,... Bm.

that is, the event A:A' happens, then A is reduced to the goal sequence B1,...,Bm which in turn begin to invoke other events, say B1:B1',...,Bn:Bn'. In this way, generally an event causes other events except the case in which a goal is unified with a clause with empty goal sequence, in this case the event causes nothing.

Let's define the causal relation among events more precisely.

"An event E, A:A', causes an event E', B:B',
if and only if B is included in the set

{ Bi | 1 =< i =< n }

where A' is a head of the clause

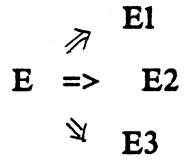
A' :- | B1,...,Bn. "

It is clear from the definition of an event that there can be no circular causal relation among events.

We denote the causal relation "E causes E'" by

E => E'

Generally an event causes more than one events.



The reflexive transitive closure of the causal relation => is denoted by ==>. By the relation ==>, an event E1 can be related to the event E2 indirectly caused by the event E1. For example, E1 => E2, E2 => E3 then E1 ==> E3 and so on.

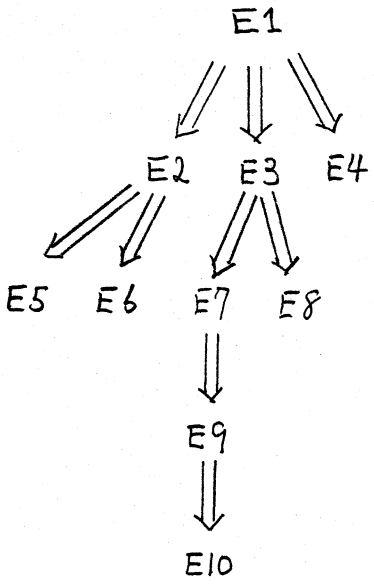
Note that the relation ==> also can be interpreted as the semi-order relation of an activation of an event. "E1 ==> E2" can be read as that an activation of an event E1 precedes an activation of an event E2.

Now we define the term "process".

"A process initiated by an event E is a chain of events
connected by the relation =>."

Given a goal A and a clause A' :- G1,...,Gn|B1,...,Bm., a process initiated by the event A:A' can be thought as the solution process of the goal A using that clause. From this point view, it is clear that the time when a process terminates is the time when the goal A is solved completely.

Since an event can cause more than one event, the chain of events (= process) looks like a tree (see figure).



The terminal nodes of the tree correspond to the events each of which is a unification between a goal and a clause with an empty goal sequence.

4. Interprocess communication

In Concurrent Prolog, interprocess communication is realized by variables logically shared among processes. A process can send a message to other processes by instantiating a variable shared among them to the message. Since a destructive assignment to a logical variable is not permitted, communication using one variable cannot be done more than once. However, in general, because there is no restriction about the number of the processes sharing a variable, the message to which one of the processes instantiates the shared variable will be sent to the rest of processes at the same time. Therefore broadcasting of a message has been realized without any additional mechanism.

Shared variables are created when, for example, a process forks to subprocesses.

```
p(X) :- | q(X,Y),r(Y?).
```

In the example above, the variable Y is shared between the processes, which are solution processes of the goal q and r respectively, and is used for communication between them.

However, as mentioned above, communication using one shared variable cannot be done more than once. Therefore in order to enable the successive communication among processes, there must be some mechanism to create a new logically shared variable dynamically. Most general method for this is the technique of the stream communication which is well known by the work of Clark and Gregory [2].

In the stream communication, a shared variable is instantiated to a data structure which contains a message and a new uninstantiated variable. In the Relational Language, a list was used for such structure.

```
[<message>|<variable>].
```

A variable contained in the structure is sent with a message from the sender to the receivers, becomes a new shared variable among processes and will be used for the next communication. Consequently as long as a process sends a message in this way, every time a message is sent, a new shared variable is created, so that the successive communication is established.

In general, the successive communication consists of two phases.

Phase 1	A shared variable is instantiated to a message.
Phase 2	A new shared variable is created.

In the phase 1, the action most essential to communication is performed. In the phase 2, what enables a next communication is performed. In the case of the stream communication, both phases are performed at the same time in the same process, the sender. However there is no reason for two phases to be performed in the same process and no restriction on the

execution order between the phase 1 and the phase 2. If we treat the two phases separately, we will be able to find several kinds of communication style based only on logically shared variables and read-only annotation. As an example, we present in this section the bounded buffer communication based on shared logical variables, which is implemented without introducing another annotation like the Relational Language. Before that, we summarize the unbounded buffer stream communication.

[The Unbounded Buffer Communication]

In the stream communication, both phases are performed at the same time in the sender of messages by instantiating a shared variable to a pair of a message and a variable. Therefore every time a sender sends a message, it gets a new "shared" variable, so that it can send a next message as soon as it sends a message. On the contrary, a receiver can read a message only after it is received and the receiver has to wait when it tries to read a message and no message is received yet. This "wait" mechanism is implemented by making the shared variable in the receiver read-only. Because there is no mechanism for inhibiting the sender to send a message, this type of communication realizes the unbounded buffer communication. Note that the essence of unbounded buffer communication is in the fact that both phases are performed in the same process, the sender of messages.

As an example of the stream communication we show the program which describes the situation where there are two communicating processes, one of which sends an integer every time the process generates it and the other prints out an integer every time the process receives it.

```
Goal:: integers(0,N) , outstream(N?).

Program-1 ::
  integers(I,[I|N]) :-                    ....    [send]
        plus(I,1,J) | integers(J,N).
  outstream([I|N]) :-                      .... [receive]
        write(I) | outstream(N?).
```

Note that "outstream" will be suspended when the variable N is not instantiated to a non-variable term, because of the condition for read-only variables. In the example above, message sendings and receivings are processed at the unification between a goal and a head of a clause. We could write the same program in more abstract level like below.

```
Goal:: integers(0,N) , outstream(N).

Program-2 ::
  integers(I,N) :-
        send(I,N,M), plus(I,1,J) | integers(J,M).
  outstream(N) :-
        receive(I,N?,M), write(I) | outstream(M).
```

In both predicates "send" and "receive", the first argument is a message, the second argument is a current communication variable and the third argument is a next communication variable. The program "send" and "receive" are:

```
send(X,[X|M],M).
receive(X,[X|M],M).
```

The advantages in using "send" and "receive" are to hide the internal structure to which the shared variable is instantiated and to modularize programs. In fact, even if we could use another data structures, say "stream(<message>,<variable>)", instead of the list "[<message>|<variable>]", the programs which have to be changed are only "send" and "receive" (new codes are shown below) and no other programs including the user programs are kept unchanged.

```
send(I,stream(I,M),M).
receive(I,stream(I,M),M).
```

On the other hand, we could say that using "send" and "receive" is to lose the simplicity of the Program-1.

[The Bounded Buffer Communication]

In the bounded buffer communication, to send a message is suppressed when messages, the number of which is equal to the size of the buffer, are kept unread in the buffer of the receiver.

From the above analysis of communication through shared variables, we can naturally find the mechanism for this kind of communication. The key idea is the separation of the actors of two phases.

The phase 1 (instantiation) is performed by the sender at the moment it send a message and the phase 2 is performed by the receiver when and only when it reads (picks up) a message from the buffer. Therefore the sender cannot send messages more than the buffer size if the receiver did not read the messages, that is, it did not generate new shared variables.

We explain the method when the buffer size is equal to two, using the previous example.

```
Goal:: integers(0,[X,Y|Z]) , outstream([X,Y|Z]).

Program::
  integers(I,N) :-
        send(I,N?,M), plus(I,1,J) | integers(J,M).
  outstream(N) :-
        receive(I,N,M), write(I) | outstream(M).
```

Note that the second argument of "send" is annotated as read-only, while in the previous example the second argument of "receive" is annotated as such. The following is a new code for "send" and "receive" programs in the bounded buffer communication.

```
send(Msg,[Msg|NewChannel],NewChannel).
receive(Msg,[Msg|NewChannel],NewChannel) :-
        wait(Msg)|update_buff(NewChannel).
```

Here again we use the list structure for implementing the stream. "wait(X)" is a system predicate which suspends when the argument "X" is not instantiated yet, and succeeds otherwise. "update_buff(X)" is a sequential Prolog program which takes a d-list as an argument and instantiates the tail variable of it to a cons cell "[P|Q]" where both "P" and "Q" are uninstantiated variables.

```
update_buff(X) :- var(X),!,X=[P|Q].
update_buff([X|Y]) :- update_buff(Y).
```

The second argument of "receive" plays a role of a buffer consisting of slots (variables) which will be filled with messages by the sender. The buffer is updated by one slot when and only when the receiver picks up a message from the buffer, so that the length of the d-list (buffer) remains the same which corresponds to the buffer size. Although the sender shares the buffer with the receiver, it can not update the buffer and all it can do is to fill empty slots with messages if there is any such slot. When the size of buffer is equal to two, the buffer looks like:

[X,Y|Z].

For the sender, the buffer looks like one of the following.

(1) [X,Y|Z]
(2) [Y|Z]
(3) Z

where "X", "Y" and "Z" are all uninstantiated variables. (1) corresponds the case in which the buffer is empty, that is, there is two empty slots and (2) corresponds to the case in which there is one room for sending a message. (3) corresponds to the case in which the buffer is full, that is, there is no room for sending a message. Because the second argument of "send" is treated as read-only, the reduction of "send" is suspended in the case (3). The figure below shows the situation where the sender tries to send three messages, "ab", "cd" and "ef" when the buffer is empty.

the receiver		the sender
[X,Y|Z]		[X,Y|Z]
	– send "ab" –	
[ab,Y|Z]		[Y|Z]
	– send "cd" –	
[ab,cd|Z]		Z
	– send "ef" is suspended	
– receive "ab" –		
[cd,P|Q]		[P|Q]
	– send "ef" –	

It is more convenient when we could parameterize the size of the buffer. Generally their usage are the following.

```
In sender     :: send(Msg,Channel?,NewChannel)

In receiver   At the first communication
              :: open(Channel,N),
                 receive(Msg,Channel,NewChannel)

                At the subsequent communications
                 receive(Msg,Channel,NewChannel)
```

"open" takes two arguments, a communication variable "Channel" and a size of a buffer "N", and it instantiates the variable "Channel" to the d-list with the first "N" arguments of it instantiated to variables. "open" is also a sequential Prolog program.

```
open(X,0) :- !.
open([X|Y],N) :- N1 is N-1,open(Y,N1).
```

The program above specifies the case in which the buffer size is more than or equal to one. Implementation of 0-Buffer communication is a little different from the above. The predicate "receive" is replaced by the following definition.

```
receive(Msg,[Msg|New],New).
```

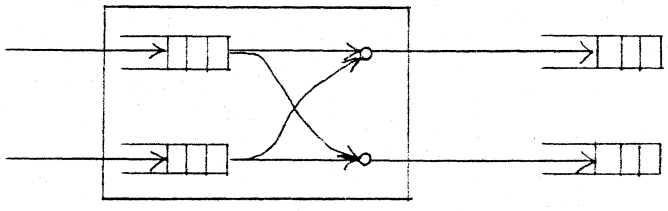
and their usage becomes:

```
In sender     :: same as above
In receiver   :: receive(Msg,Channel,NewChannel),wait(Msg)
```

The bounded buffer communication is very important when there are several processes, each of which produces or consumes data in different speed. Suppose that, in the example above, the rate of integer generation in "integers" is much greater than that of data consumption in "outstream", in such case if we use the unbounded buffer communication between two processes, the huge amount of unprocessed integers will be produced. The bounded buffer communication is a simple and efficient method to control and combine processes having different rate of data producing or consuming by controlling the production of data according to the consumption of them.

As an example of the application of this bounded buffer communication, we can define a 2 x 2 communication switch which has two input ports and two output ports. It can receive inputs from two ports and sends them to the output port which has at least one empty slot. If both ports are not available, the "switch" is suspended.

```
switch2x2(In1,In2,Out1,Out2) :-
     receive(M,In1,Ins1)&send(M,Out1,Outs1) | switch2x2(Ins1,In2,Outs1,Out2).
switch2x2(In1,In2,Out1,Out2) :-
     receive(M,In2,Ins2)&send(M,Out1,Outs1) | switch2x2(In1,Ins2,Outs1,Out2).
switch2x2(In1,In2,Out1,Out2) :-
     receive(M,In1,Ins1)&send(M,Out2,Outs2) | switch2x2(Ins1,In2,Out1,Outs2).
switch2x2(In1,In2,Out1,Out2) :-
     receive(M,In2,Ins2)&send(M,Out2,Outs2) | switch2x2(In1,Ins2,Out1,Outs2).
```



5. Incomplete Message

As in the actor formalism, Concurrent Prolog is a model of the parallel computation and provides a communication methods through shared variables. A message will be sent by instantiating the shared variables. A message which contains a variable is called an incomplete message [5]. It makes a new variable shared by the sender and the receiver of the

message, that is, it creates a new communication channel. It means that a communication channel can be made dynamically and it can be sent to other processes also.

The concept of an incomplete message is a large programming paradigm which includes the basic communication mechanism between processes, so-called pipeline processing on stream data, and yields new features of Concurrent Prolog. The close analysis of this concept is described in the paper of Shapiro and Takeuchi [5].

In this section, we review the key features of this concept according to the paper of Shapiro and Takeuchi [5].

(1) [Stream] Once a variable is instantiated, it will never be rewritten except the case where the whole goals fail. Therefore it can not be used as a communication channel in the next message passing phase. In order to enable subsequent communication, in the stream communication generally a shared variable is instantiated to a list of a message and a variable which will be used in a next communication. In this sense, the stream communication is one of the examples of incomplete messages and provides a basic communication mechanism in Concurrent Prolog.

(2) [Pipeline] In addition, incomplete messages make it possible to process partially obtained data in a pipeline style. Although pipeline processing on stream data is a new concept of programming languages, it is included naturally in the paradigm of the partially defined message. In some sense, usual message passing can be seen as a kind of pipeline processing on a sequence of commands generated incrementally.

(3) [Response] When a process sends a message which requires a response, the response can not be sent through the same shared variable, since logical variables are single-assignment. The technique of the incomplete messages is also useful in this case, in which the sender sends a message that contains an uninstantiated variable, and then examines that variable in a read-only mode, which causes it to suspend until this variable gets instantiated to the response by the recipient of the

message. However this is different from the examples above, because the process which instantiates a shared variable is the receiver of the message. In this case, once a message is sent to a process, the sender can run independently whether the receiver returns the response as long as the sender need not to refer to the response. When the sender needs the response it is forced to wait until it will be instantiated. This behavior associated to a shared variable used in a response takes an advantage in writing a monitor of shared resources and highly reduces the overhead on the resource manager because the manager will never be locked and the request will never be refused.

6. New Features of Concurrent Prolog

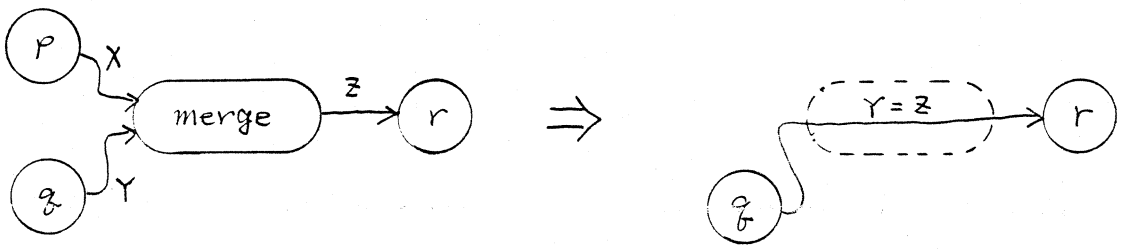
In this section, we explain the another feature of the Concurrent Prolog not available in other concurrent programming languages.

The interprocess communication based on shared variables is not new method and has been implemented generally by sharing physical memory cells. The difference between the communication by the shared variables of Concurrent Prolog and that of traditional languages is the highly abstracted level of shared object. In traditional languages, the objects shared are physical objects such as memory cells or global variables. On the contrary, in Concurrent Prolog, the objects shared are highly abstracted logical variables which can be objects of the unification operation, a very high level operation. Because of this high level abstraction, Concurrent Prolog can express very high level communication style among parallel processes in a simple way, that is, unifying two communication channels.

The well-known "merge" program is an example of this feature.

```
merge([A|X],Y,[A|Z]) :- | merge(X?,Y,Z).
merge(X,[A|Y],[A|Z]) :- | merge(X,Y?,Z).
merge([],Y,Y).
merge(X,[],X).

Goal:: p(X),q(Y),merge(X?,Y?,Z),r(Z?)
```

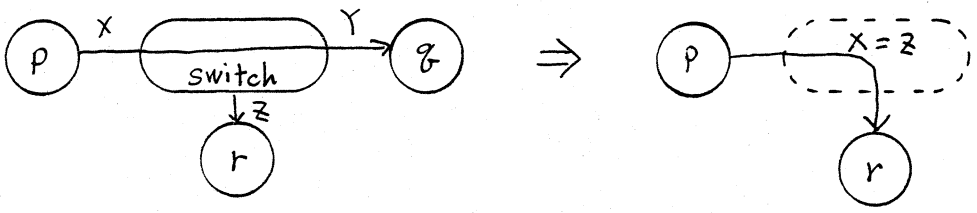
This program merges two input streams into one stream. The first two clauses are used for this purpose. The rest two clauses describe the situation, where one of the input stream (say "X") reached the end, and the remaining stream ("Y") is unified with the output stream ("Z"). After this unification, data on the remaining stream ("Y") are sent to the output stream ("Z") without any relay, because the input stream and the output stream are logically the same. The important point is that this change of the data flow can be performed only by the unification and that both the sender and the receiver never know the change of data flow (Figure).



The next program shows another example.

```
switch([on|X],[],X).
switch([A|X],[A|Y],Z) :- | switch(X?,Y,Z).

Goal:: p(X), switch(X?,Y,Z), q(Y), r(Z).
```

"switch" takes three arguments. The first argument is the input stream and the second and the third are the output streams. "switch" program keeps the connection between the input stream and the second argument until it will find the "on" message in the input stream. When "switch" receives it, it changes the connection and thereafter it will pass input data to the third argument. Here again the important point is that the the data flow can be changed directly by the unification and it is hidden from both the sender and receivers (figure).



These two examples demonstrate the new feature of interprocess communication in Concurrent Prolog. Other powerful examples are presented in the paper [5].

7. Conclusion

In this paper we present the computation model of Concurrent Prolog and explain mainly the interprocess communication based on the shared logical variables. 1) From the close analysis of the stream communication, we derived the mechanism for implementing the bounded buffer communication only by the read-only annotation. 2) We have shown briefly the basic programming paradigm "incomplete messages" as a source of the powerful programming technique. 3) We have shown the new features of Concurrent Prolog programming which originate from the logical power of the unification.

8. Acknowledgement

We thank E. Shapiro for his many helpful insights and discussion. We would also like to thank Kazuhiro Fuchi, Director of ICOT Research Center and all the other members of ICOT, both for help with this research and for providing a stimulating place in which to work.

9. References

[1] E.Y.Shapiro: A Subset of Concurrent Prolog and Its Interpreter, ICOT Technical Report TR-003 (1983).

[2] K.L.Clark, S.Gregory: A Relational Language for Parallel Programming, Proceedings of the ACM Conference on Functional Programming Languages and Computer Architecture (1981).

[3] C.Hewitt: Viewing Control Structures as Patterns of Passing Messages, Artificial Intelligence 8 (1977).

[4] S.A.Ward, R.H. Halstead: A Syntactic Theory of Message Passing, JACM Vol.27, No.2 (1980)

[5] E.Shapiro, A.Takeuchi: Object Oriented Programming in Concurrent Prolog, New Generation Computing Vol.1, No.1 (1983)

Intelligent Backtracking for Automated Deduction in FOL

Stanisław Matwin
Department of Computer Science, University of Ottawa, Ontario, Canada

Tomasz Pietrzykowski
School of Computer Science, Acadia University, Nova Scotia, Canada

Abstract

An "intelligent" backtracking algorithm for depth-first search of the solution space generated during linear resolution in fol has been designed. It inspects only a small portion of the total solution space, which consists of special graphs representing the deductive structure of the proof. These graphs are generalization of AND/OR trees. Our (partially) complete search algorithm has natural potential for parallel implementation. However, it may generate redundant refutations; it seems that this is the effect of the prevailing design objective, which in our case was completeness of the method.

A preliminary estimate of the efficiency of the algorithm has been carried out. It indicates exponential speed-up over the worst case of linear backtracking.

An implementation (3000 lines of PASCAL code, under CMS) has now been completed. That allows us to experiment with the algorithm and investigate certain open questions.

1. Introduction

Many researchers working in Artificial Intelligence and its applications agree that an efficient backtracking mechanism will drastically expand applicability of Logic Programming ([Warren et al 77], [Pereira and Porto 80], [Nau 82], [Stallman and Sussman 77]). One such algorithm has been designed by M. Bruynooghe [78] and L.M. Pereira [79], [80]. This paper presents an alternative and different approach. Our method is based on a graph-based, depth-first proof procedure [Cox and Pietrzykowski 81]. The basic notion, on which this algorithm is based, is the plan: a directed graph, representing deductive structure of the proof. The plan is a natural generalization of AND/OR trees. The unifications, generated during the proof, are kept in a separate graph structure, called the graph of constraints. In this way, even if backtracking along a particular path of the plan does not lead to a solution and this path will have to be re-generated, there is no need to re-generate the unifications obtained along that path.

Furthermore, our method is applicable to general first order logic, without being restricted to Horn clauses. Also, as it will be demonstrated later, intelligent plan-based deduction has natural potential for a parallel implementation.

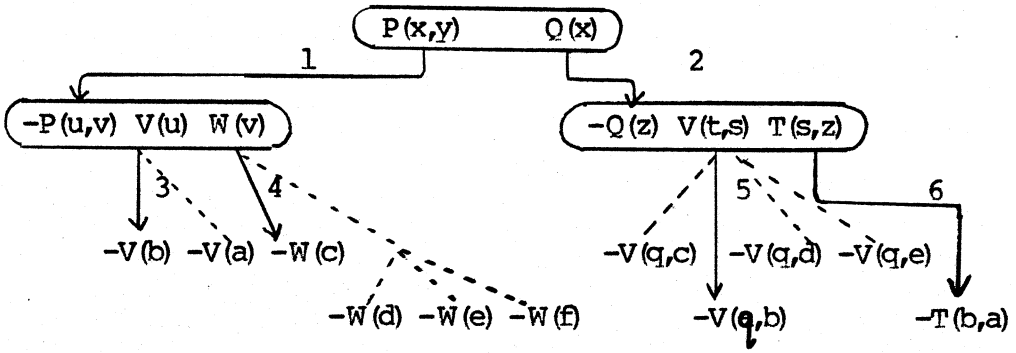
Finally, it has to be emphasized that the prevailing design criteria of our algorithm was completeness of search of the search-space. This has been achieved, and the proof of (partial) completeness has been obtained [Matwin and Pietrzykowski 83]. However, a price which had to be paid is redundancy (i.e. the same solution may be obtained more than once). Bruynooghe-Pereira method does not suffer from this deficiency, but then it is not certain that their solution is a complete one.

. Operation of the Intelligent Backtracking System.

Before introducing the algorithm and a more complete example, let us illustrate he difference between "exhaustive" and "intelligent" backtracking using a very imple case. Assume that the following set of clauses is given:

```
 P(x) Q(x).                    -Q(z) S(t,s) T(s,z).
-P(u,v) V(u) W(v).             -S(q,b).    -S(q,c)
-V(b).    -V(a).               -S(q,d).    -S(q,e)
-W(c).    -W(d).               -T(b,a).
-W(e).    -W(f).
```

learly, with the left-to-right "reduction" (although "expansion" seems to be more dequate term) policy, the following plan, which in this case is just and AND/OR ree, is obtained:



he continuous lines represent the AND arcs, the dotted are the OR arcs. bviously, in this tree there is a clash between constant b, generated by arc 3, and onstant a, generated by arc 6. One look at the plan convinces us that arc 3 is he culprit, and that a reduction following its alternative remedies the problem. owever, exhaustive backtracking will perform 33 reductions [3*2 + 3*((4*2) + 1) = 3] before generating the solution. The reason for that is the fact that all the lternatives between the arcs 3 and 6 involved in the conflict are tried by xhaustive bactracking. Our method is different: it only tries 6, 3 and the lternatives lying above them. In this case, one reduction replacing 3 with its lternative will do the job. In a reasonably balanced AND/OR tree, the number of lternative deductions obtainable in between two modes is of the order exponential rt the height of the tree. Therefore, a method which operates only above the lashes will be exponentially faster than the worst-case behaviour of exhaustive acktracking discussed here.

We shall now proceed with a more thorough discussion of our method, beginning ith the underlying notions and concepts.
The basic structure, involved in the algorithm is the plan. By a plan we nderstand a directed graph, nodes of which represent variants of clauses. One of he nodes, referred to as TOP, represents the clause to be proven. Arcs of the lan connect pairs of literals, belonging to individual nodes. Each two literals, efining ann arc, are unifiable and of opposite sign. There are two types of arcs: UB arcs and RED arcs (as proven in [Cox and Pietrzykowski 81], those two rules rovide a complete set). Informally speaking, SUB arcs point "downwards" in the lan, while RED arcs point "upwards". Each node, except the TOP, is entered by xactly one SUB arc (and, possibly, by zero or more RED arcs). The literal within

a node, pointed to by a SUB arc, is the key of this node. Each other literal of this node is called a goal. A goal is called closed if there is an arc, originating in this goal, otherwise the goal is an open one.

With each goal of the plan we associate a set of arcs, called the set of potentials. They are the arcs which could have been generated instead of the one actually created. Let us notice that, if the plan is a tree, then the initial value of all potentials represents all the OR arcs. In any case, this initial value is static information.

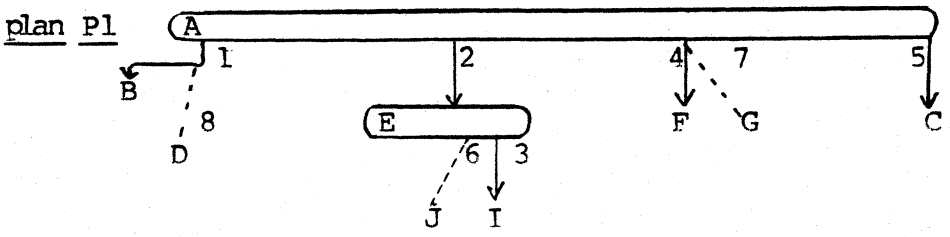
As mentioned before, the information gathered as result of unification is kept separately, in a special data structure called the graph of constraints. This graph reflects the history of unifications which have taken place in the proof during its progress. A node of the graph of constraints, called a constraint, represents the information about the bindings which have ben imposed on a variable during the history of proof. Therefore, presence of two different constants in a constraint is an indication of a clash. This clash is then mapped on the plan. Each minimal set of plan arcs such that its removal annihilates the clash is referred to as a conflict. The conflict set is the set of all such conflicts for a given plan. In some situations, even though the conflict set is empty, we want to create an artificial conflict set, in order to assure completeness. Artificial conflict set contains all the arcs entering unit clauses, and all the reduction arcs.

Finally, our method introduces two other notions, motivated by memory management problems. The algorithm uses a repository of plans, accompanied by their graphs of constraints and conflict sets. This repository, called the store resides on disk, and plans are fetched from it and added to it. There is always one plan being operated on: it is called the table plan (or simply the table).

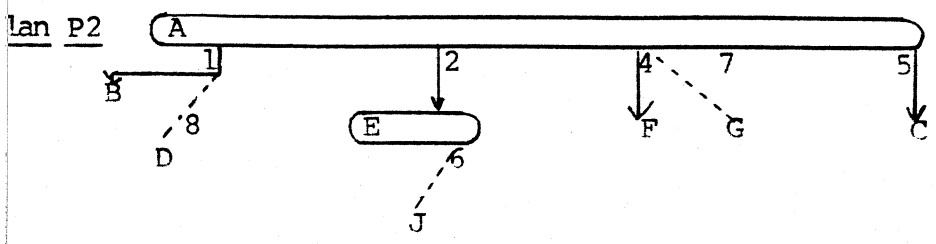
With this background, we can now follow the operation of our algorithm on the following set of clauses:

A. P(x) Q(y) R(x,y) V(a,x) = TOP
B. -P(a)
C. -V(t,y)
D. -P(c)
E. -Q(w) V(v,w)
F. -R(z,z)
G. -R(u,v) S(u)
H. -S(a)
I. -V(b,b)
J. -V(c,c)

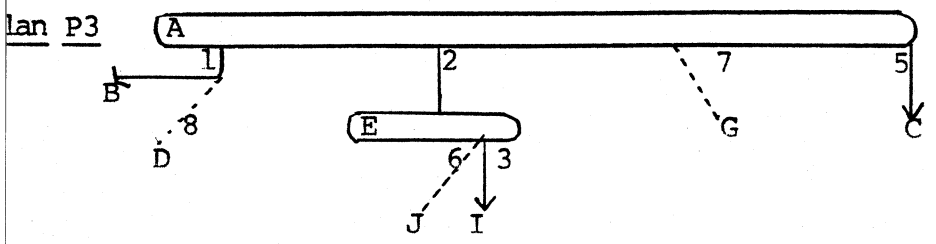
Initially, the store is empty. Clause A is chosen as the TOP and a single-node plan consisting of A is generated. Since it is not closed, it will be further developed until either a closed plan is obtained or a non-empty conflict set is generated. In our case, we get the following plan:



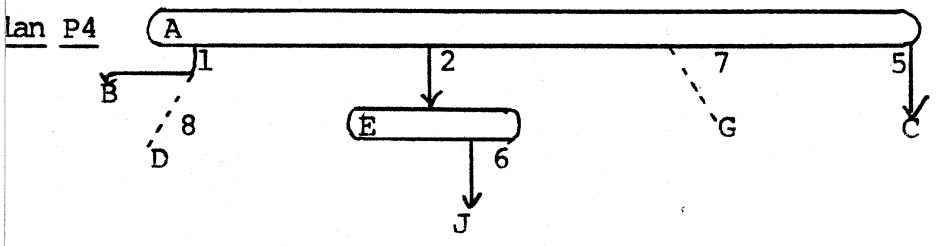
s conflict set is (3, 1 ∧ 5; 4). Suppose that 3 is chosen for removal; the open lan P2 is obtained and placed in the store:



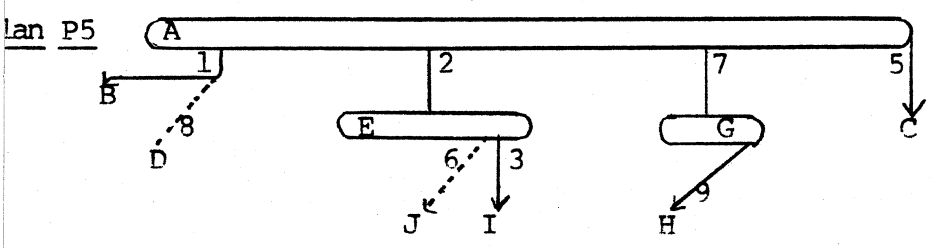
he conflict on the table is now (1 ∧ 5, 4). If 1 ∧ 5 is chosen, prunning annihilates he plan, as 5 has no potential and A is the TOP. With the choice of 4, open plan 3 is obtained and placed in store:



ince there are no more conflicts on the table, one of the store plans (suppose it P2) is placed on the table. Potential 6 is realized as an arc, which leads to a onflict set (1 ∧ 5, 6, 4) on the table. Since choice of either 6 or 1 ∧ 5 leads owhere, suppose that 4 is chosen and P4 is sent to store.

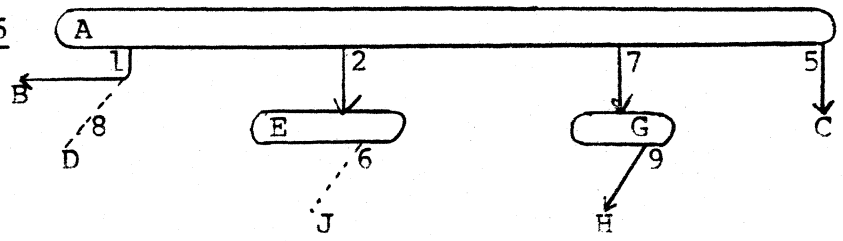


s the conflict set is again empty, one of the store plans P4 and P3 is placed on he table. Assume P3 is chosen; the only open goal is closed with its potential 7 nd a solution is obtained:



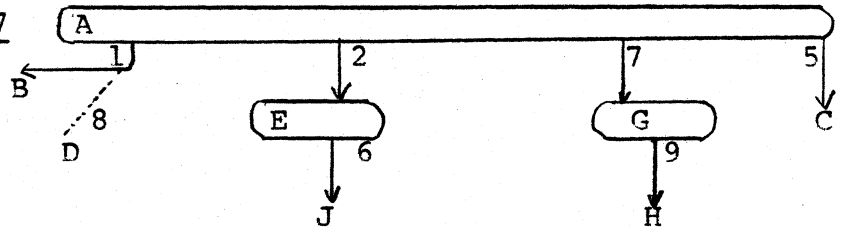
rtificial conflict set of P5 is (1, 3, 5, 9). Removal of 5 and 9 leads nowhere as here are no potentials between these arcs and the top. Replacement of 1 by 8 is lso unproductive (the reader will easily see why). This leaves us with 3 as a easonable candidate for removal, which results in plan P6.

plan P6



The only plans in store are now P4 and P6. With the choice of P6, potential J is realized and we get solution P7:

plan P7



Since all attempts of obtaining new plans from its artificial onflict set fail, the only remaining store plan, P4, is placed on the table. Its potential 7 and then arc 9 are realized, which gives a redundant solution identical to P7. The store is now empty and the algorithm terminates.

We have proven elsewhere [Matwin, Pietrzykowski 83] that when our algorithm terminates, it generates all the existing proofs (partial completeness).

3. Further Inhancements of the Algorithm

There are at least three directions of further research, leading to potentially interesting enhancements of the algorithm.

1. Different strategies for nondeterminism. A number of nondeterministic choices is involved in the algorithm. Two types of such choices were mentioned in the brief description in the preceding section: choice of the plan from the store to be placed on the table, and choice of a conflict from the set of conflicts. It is not clear, at this stage, what are the right criteria for these choices. This is particularly important when the objective is to find a proof, rather than all the possible proofs. It seems that in this case the right strategy may bring about significant increase in efficiency.

2. Applicability of the algorithm in the domain of expert systems. The researchers in expert systems point out that a method of limiting the search space is of great importance for implementation of practical systems [Nau 83]. The early work of [Stallman and Sussman 77] bears a good deal of resemblance to our method, although their approach is less general. System ARS, reported in [Stallman and Sussman 77] implements a method of dependency directed backtracking, tailored to the particular environment of algebraic relationships encountered in the analysis of electric circuits. Therefore it seems that a method like ours may be productive, particularly in case of expert systems using fol or its derivatives [Skuce 83] to represent knowledge bases.

3. Distributed implementation. Since no ordering of conflicts in within the

MATW.

conflict set is assumed, an interesting parallel implementation seems possible. It will involve a number of processors, each of which would remove a conflict, carry out the necessary pruming (if any) and develop the plan. The result of development is placed in store, ready to be picked up by another processor. The whole system stops when the store becomes empty. Such a parallel, distributed implementation seems to be feasible. Let us notice that the similar approach to Bruynooghe-Pereira method would not work, since their algorithm specifically orders the conflicts, which in turn allows them to avoid the redundancy problem.

eferences

Bruynooghe 78] Bruynooghe, M., Intelligent Backtracking for an Interpreter of Horn Clause Logic Programs, Procs. of Colloquim on Mathematical Logic in Programming, Salgotarjan, Hungary, 1978.

Bruynooghe and Perreira 81] Bruynooghe, M., Pereira, L.M., revision of Top-Down Logical Reasoning Through Intelligent Backtracking, res. Report of KUL and CIUNL, 1981.

Cox and Pietrzykowski 81] Cox, P., Pietrzykowski, T., Deduction Plans: A Basis for Intelligent Backtracking, IEEE PAMI, Jan. 1981.

Matwin and Pietrzykowski 82] Matwin, S., Pietrzykowski, T., Exponential Improvement of Exhaustive Backtracking: Data Structure and Implementation, Procs. of CADE-6, 1982.

Matwin and Pietrzykowski 83] Matwin, S., Pietrzykowski, T., Intelligent Backtracking in Plan-Based Deduction, Submitted to IEEE PAMI.

Nau 83] Nau, D.S., Expert Computer Systems, IEEE Computer, Feb. 1983.

Pereira 79] Pereira, L.M., Backtracking Intelligently in AND/OR Trees, Research Report, CIUNL 1979.

Pereira and Porto 80] Pereira, L.M., Porto, A., Selective Backtracking for Logic Programs, Procs. of CADE-5, 1980.

Pietrzykowski and Matwin 82] Pietrzykowski, T., Matwin, S., Exponential Improvement of Exhaustive Backtracking: A Strategy for Plan-Based Deduction, Proc. of CADE-6, 1982.

Skuce, 83] Skuce, D., KNOWLOG, Submitted to IEEE Computer.

Stallman, R.M. and Sussman 77] Stallman, R.M., Sussman, G.J., Forward Reasoning and Dependency-Directed Backtracking in a System for Computer-aided Circuit Analysis, Artificial Intelligence, 1977.

Warren et al 1977] Warren, D.H.D., Pereira, L.M. Pereira, F.,. PROLOG - The language and its implementation compared to LISP, ACM SIGPLAN, Aug. 1977.

LOGICAL ACTION SYSTEMS

Antonio Porto

Departamento de Informatica
Universidade Nova de Lisboa
2825 Monte da Caparica
Portugal

ABSTRACT.

Logic programming is being hailed by many people as a good way towards a side-effect-free programming style. On the other hand, talking about temporal effects or **actions** is the natural way of viewing many common computational phenomena, such as input/output or database update operations.

The purpose of this paper is to introduce some common ground in the form of **logical action systems**, a framework for dealing with actions that has its roots in logic programming. Programs consist of rules for action reduction; rules have preconditions as Prolog-like goal expressions and define state transitions in the form of deletion and/or creation of assertions. **Concurrency** of actions is supported. **Abstract data types** can be defined.

INTRODUCTION.

Despite the defense by many people of a side-effect-free programming style, as in a 'pure' logic programming system, the fact remains that many common computational phenomena are not naturally expressed without resorting to the notions of action and state transition.

Rather than considering actions as impure side-effects arising within a pure logic computation, why not invert the situation and consider logic computations as a normal side-effect of action systems?

We will put forward a proposal for a language in which to describe **logical action systems (LAS)**, providing a clear link between actions and normal logic programming.

Logic and **unification** are still the basis on top of which LAS are conceived; however, actions are clearly separated from purely deductive goals.

The language can be seen as yet another proposal for expressing **concurrency**.

We will begin by exposing the main ideas behind LAS, and then move on to an **object-oriented** approach with **abstract data type** definitions.

ACTIONS.

Actions take place on some world, modifying it. Between occurrences of actions we can refer to the **state** of the world.

We represent a world in two parts, each one of them a logic program:

(1) the **rules of the world**, defining relations that are not bound to change in time ;
(2) the **state of the world**, containing assertions that may change in time as the result of actions performed on the world.

Let us look at an example. (Edinburgh Prolog syntax will be used, except for clause functor.)

Consider a blocks world.

The world rules would contain definitions such as :

```
tower([B]) <- on(B,floor).

tower(B1.B2.Bn) <- on(B1,B2), tower(B2.Bn).
```

A particular world state would have assertions such as :

```
on(a,b).

on(b,floor).
```

At any time between actions it is possible to evaluate a goal expression against the world, using the rules and the state as a joint logic program.

In this example, one could evaluate the goal

```
<- tower(X).
```

that would yield the solutions

```
X=[b] ; X=[a,b] .
```

The action of moving 'a' to the 'floor' would replace the assertion 'on(a,b)' by 'on(a,floor)'. As a consequence we would have a new world state, and the same goal '<-tower(X)' would now produce the solutions

```
X=[a] ; X=[b] .
```

In general, an action will consist of a number of action steps (possibly infinite).

Each action step will result in a change of state, consisting of falsifying (deleting) some assertions of the previous state and/or making true (creating) some new assertions.

The specification of an action step is an **action rule**. It is made up of two parts : the **action reduction** and the **state conditions**.

The **action reduction** defines what new actions the action reduces to, by virtue of the step.

The state conditions are the preconditions and the state transitions.

Preconditions are goals that are evaluated against the world in its current state.

State transitions tell what assertions must be deleted from, and what new ones added to, the current state to get the new state.

For example, the notion that the action of moving A to B can be accomplished if nothing is on top of A and B, and as a result A ceases to be on top of whatever it was before to be on top of B, can be described by the action rule :

```
move(A,B)  <=  not on(_,A),
               not on(_,B),
               on(A,_) -> on(A,B).
```

ACTION REDUCTION.

When an action reduces to void (is finished), as in the preceding example, the action reduction part of the rule is just the action - the rule head.

In general, an action reduces to other actions. The action reduction part of a rule is then of the form

A -> NA

where A is some action (the rule head), and NA is an action expression referring to the new actions.

How can actions relate to one another to form an action expression? We find that we need two connectives: parallel and sequence.

Two actions in sequence are denoted by 'A,B' , meaning the second action (B) can only take place after the first one (A) is finished.

Two parallel actions, written 'A/B' , may take place with no time constraints on one another.

An action expression is recursively constructed from atomic actions and the parallel and sequence connectives. Relative precedence between these is such that 'A/B,C' is the same as '(A/B),C' .

In an action system there is always an action expression evolving in time and denoting at each moment the actions that are to be carried out in the world. We call it the agenda. For every action in the agenda that is ready to be carried out (for example, A1 and B1 in (A1,A2)/(B1,B2)), the system tries to apply an action step.

The action reduction involved in a step is like a rewrite rule for the ready action in the agenda, keeping the overall structure of this action expression.

Thus, if we have the agenda

A,B

and the action reduction

A -> A1/A2

is performed, the agenda becomes

A1/A2,B

meaning that after A1 and A2 are both finished (having done so independently of one another) B is ready to take place.

Actions occur in time and time always runs forward, so there is no question of backtracking over action steps. If an action is required and no rule for that action applies in the current state, it just means that the action must **remain** in the agenda **waiting** for the right conditions to appear (when some other action changes the state to that effect). This eventually entails the well-known phenomena of deadlock and starvation.

STATE TRANSITIONS.

State transitions inside an action rule may be of three types :

(1)	-> A	assertion A is created ;
(2)	A ->	assertion A is deleted ;
(3)	A -> NA	assertion A is deleted and assertion NA is created.

Of course a type 3 transition is no more than a type 1 and a type 2 put together, but it makes for a more clear reading of the rule, especially if A and NA are for the **same predicate**. In this case, a compiler or interpreter can easily translate the transition into simple **assignments** on the changing arguments, with considerable speed-up over deletion and creation.

RULE EVALUATION.

Each rule is associated with a **single** action (the rule head), so a ready action in the agenda can **efficiently** trigger its own rules, much as Prolog goals trigger their clauses.

Rule evaluation begins with **unification** of the ready action with the rule head.

If there are any type 2 or type 3 transitions in the rule, their left-hand side is regarded as a **goal** to be matched against an **assertion** in the current world state. All **precondition goals** together with these **transition goals**, in the order in which they appear in the rule, form a Prolog goal expression that is evaluated. If a solution is found, then the rule applies, and the transitions are carried out, deleting the assertions that matched the transition goals for the solution found.

The action is replaced in the agenda by the new action expression it reduced to, with the obvious simplifications when this is void.

There may be several rules for a given action. Rules should be tried in the **order** in which they appear in the program. This provides a simple, elegant form of **if-then-else**.

For example, the complete definition for the 'move' action in the blocks world might be :

```
move(A,floor) <=  not on(_,A),
                  on(A,_) -> on(A,floor).

move(A,B) <=  not on(_,A),
              not on(_,B),
              on(A,_) -> on(A,B).
```

giving preference to 'move's to the 'floor', if destination is unspecified.

SYNCHRONIZATION.

What is usually referred to as **process synchronization** is achieved in a LAS by the combined effect of the **sequence** connective and **state transitions** seen by the "processes".

Imagine a single cell buffer, defined by the following actions :

```
put(X)  <=  empty -> with(X).

get(X)  <=  with(X) -> empty.
```

A 'put' action will only be accomplished if the buffer is empty, and, conversely, a 'get' action can only be carried out if the buffer contains something. So actions sequenced after a 'put' will eventually have to wait for the 'get' of a previous token put in the buffer, and actions sequenced after a 'get' will eventually have to wait for the 'put' of the corresponding token, thus achieving synchronization of the two "processes" using the buffer.

CONCURRENCY.

Parallel actions are performed **concurrently**. So it is crucial that any sound implementation of the system be able to guarantee, just before performing a state transition, that the preconditions of the rule still apply. In other words, care must be taken with regard to state transitions occurring during the evaluation of a rule. A number of techniques exist for tackling this problem, depending on the actual hardware, but their discussion is outside the scope of this paper.

Let us look at an implementation of a queue in terms of its

accessing actions 'put' and 'set'. The queue itself is implemented as a difference-list Q-T via an assertion 'q(Q,T)', acted upon by 'put' and 'set' :

```
put(X)  <=  q(Q,X.T) -> q(Q,T).

set(X) <=  q(Q,T) -> q(NQ,T),
           nonvar(Q),
           Q=(X.NQ).
```

This queue "process" puts in a list all elements X for which a 'put(X)' action is requested, in the order in which these actions are performed (since they can always be executed, apart from simultaneity with 'set' actions, this will be the order in which they become ready in the agenda).

This is in contrast to other formalisms, such as Concurrent Prolog [Shapiro 83], that deal with explicit **streams** and thus require the explicit **merge** of the various input streams to a queue.

Let us look at another classic example of concurrent programming, the problem of the dining philosophers. Five philosophers are seated around a table, with a fork between each two of them (five in all) and a central bowl of spagethi. Whenever a philosopher stops thinking because he gets hungry, he must pick up the two forks on his left and right and begin eating until satisfied, letting then down the two forks and resuming his thinking.

Philosophers - p1, p2, p3, p4, p5

Forks - f1, f2, f3, f4, f5

World rules :

```
forks(p1,f1,f2).
forks(p2,f2,f3).
forks(p3,f3,f4).
forks(p4,f4,f5).
forks(p5,f5,f1).
```

Initial world state:

```
down(f1).
down(f2).
down(f3).
down(f4).
down(f5).
```

Initial agenda :

```
-> thinking(p1) / thinking(p2) / thinking(p3) /
   thinking(p4) / thinking(p5).
```

Action rules :

```
thinking(X) -> hungry(X).

hungry(X) -> eating(X)  <=
             forks(X,L,R),
             down(L) -> with(X,L),
             down(R) -> with(X,R).

hungry(X) -> wants_fork(X,R)  <=
             forks(X,L,R),
             down(L) -> with(X,L).

hungry(X) -> wants_fork(X,L)  <=
             forks(X,L,R),
             down(R) -> with(X,R).

wants_fork(X,F) -> eating(X)  <=
                   down(F) -> with(X,F).

eating(X) -> thinking(X)  <=
             with(X,L) -> down(L),
             with(X,R) -> down(R).
```

Some comments are due.

The first and last rule, of course, do not show any details about when to set hungry or when to stop eating. For an actual simulation we should provide adequate mechanisms, say a random time lapse generator.

It is important to note that, in the last rule, the two 'with' transition goals must match two **distinct** assertions and not the same one. Type 2 or type 3 transitions inside the same rule always refer to distinct assertions, for it would make no sense to specify two deletions of a single assertion.

The aforementioned if-then-else effect of rule evaluation implies that, when a philosopher gets hungry and both his two forks are available, he will pick them up **simultaneously**. This fact entails that there is no deadlock or starvation if the system starts from a non-deadlock initial state, as can be easily proved. What happens is a transfer of deadlock/starvation monitoring to the underlying execution mechanism of LAS, when concurrently trying to apply action rules. We are in fact assuming that no ready action is indefinitely postponed if conditions indefinitely exist for its reduction.

ABSTRACT DATA TYPES.

One of the nice extensions of the action system presented so far is the introduction of **abstract data type** (ADT) definitions. This provides a much needed **modularity**, in the form of **local** assertions that cannot be globally accessed, and

are manipulated only by the actions interfacing the ADT object with the rest of the system.

A definition of a queue ADT might be the following :

```
type queue.

put(X)  <=  q(Q,X,T) -> q(Q,T).

get(X)  <=  q(Q,T) -> q(NQ,T),
            nonvar(Q),
            Q=(X.NQ).

#

q(X,X).

*
```

The first part of an ADT definition, until the character '#', defines the external actions that may be used to access an object of the given type.
In this example we have the previously defined 'put' and 'get' actions.

The second part of the definition, until the character '*', defines internal rules and assertions, that cannot be accessed from the outside.
The assertions correspond to the initial state of an object, when it is created. There can also exist, in the second part of an ADT definition, an initial agenda '->A' to be launched upon creation of an object.
In the preceding example the initial state is an empty queue, as defined by the assertion 'q(X,X)', and there are no internal actions or initial agenda. The assertion being local, it won't be "seen" by any outside goal 'q(_,_)'.

Having defined an ADT, we must have means to create and kill objects of that type. We use the system-defined actions

```
create(Object,Type)
```

and

```
kill(Object) .
```

Now actions directed at an object must refer to it. We use the notation

```
Object:Action
```

for that kind of actions.

Let us look at a more complex example, a definition of a video terminal. The keyboard is scanned to get characters

typed in it. In the local mode each character is output on the screen; however, if the character is a 'send', character output is diverted to the outside of the terminal, until the character 'eot' is found, in which case there is a switching back to local mode. The terminal can be accessed from the outside through a 'put(X)' action, resulting in character X being displayed in the screen.

This ADT has three parameters, 'Keyboard', 'Screen' and 'Out', which are supposed to be ADT objects themselves. 'Keyboard' is supposed to be accessed through a 'get' action, while 'Screen' and 'Out' through a 'put'.

```
type terminal(Keyboard,Screen,Out).

put(X) ->  Screen:put(X).

#

terminal(X) -> select(X) / Keyboard:get(NX), terminal(NX).

select(send)  <=  local -> out.
select(eot)  <=  out -> local.
select(X) -> Screen:put(X)  <=  local.
select(X) -> Out:put(X)  <=  out.

local.

-> Keyboard:get(X), terminal(X).

*
```

External access is permitted only through a 'put' action. 'terminal' and 'select' are **internal** actions - 'terminal' performs the endless loop of getting characters from the keyboard and processing them; 'select' does this processing.

The **initial state** is local mode, and the terminal activity is started by getting a character from the keyboard and entering the loop.

A terminal, being accessed through a 'put', can serve as the 'Out' object of another terminal. We can for example link two terminals together :

```
-> create(T1,terminal(k1,s1,T2)) /
   create(T2,terminal(k2,s2,T1)).
```

DEDUCTION AS ACTION.

We tackle here the problem of treating as an action the work of a Prolog interpreter while trying to execute a goal. Keep in mind that backtracking is "backward" as far as the object language goes, but is "forward" as regards the temporal activity (action) of the interpreter.

A drawback of Prolog is revealed when we want to keep track of different solutions to a goal while getting them on demand, along with some other computation. The problem lies in the fact that we are using a single interpreter, and backtracking, that is needed locally to provide the various solutions, is only available as a global operation.

"Metapredicates" like 'setof' or 'all' only give the whole set of solutions to a goal, and cannot be used in the desired coroutined way.

A way out in the framework of LAS is to have the Prolog interpreter defined as an ADT, accessible through the action of producing the next solution to a goal.

We can then create an instance of the interpreter by the action

create(I,interpreter(G,T))

where G is the goal expression to be interpreted, and T is the term whose instances we seek.

Transporting the name I of this particular interpreter we can then get on demand the next solution, with the action

I:next(X) .

As a result, X is bound in the action environment to a copy of the next instance of T found by I to be a solution for G (T and G will remain unbound in the action environment).

This is to say that several interpreter objects are truly decoupled in the sense that they don't share their binding environments. Some more thought should be given to this theme of sharing versus copy, in the context of LAS.

There remains the problem of failure. The action 'next' is always carried out, but it should produce information regarding its outcome, that can be used in the action context.

Maybe this type of actions should really be used as a logical goal, with associated meaning the truth or falsehood of the possibility of performing the action.

We can turn this into a general property of actions, with the assumption that the default boolean value of an action is true when the action is normally finished, and false if unfinished when the special action 'abort' is carried out, making "impossible" the whole action expression (agenda) where it occurs (it becomes empty).

Remember that using ADTs one has distinct agendas for each object, and thus 'abort' can be used in a modular rather than global way. One can, for example, implement a Unix-like shell using the 'abort' action triggered by the control_C trap to abort execution of the current command !

```
type shell(Input,Command_interpreter).

#

commands(C) -> Input:get(NC) / Command_interpreter:C ,
               commands(NC).

control_C_trap(C) -> Input:get_ahead(NC) / check(C) ,
                     control_C_trap(NC).

check(^C) -> Command_interpreter:abort.
check(_) -> .

-> Input:get(C),
   commands(C) / control_C_trap(C).

*
```

REFERENCE.

[Shapiro 83]

Ehud Shapiro.
A Subset of Concurrent Prolog and its interpreter.
The Weizmann Institute of Science.

Issues in Developing Expert Systems

Jack Minker

Department of Computer Science

University of Maryland

College Park, Maryland 20742

1. Introduction

The purpose of this note is to provide a brief overview of the field of expert systems, and to set forth some issues to be discussed in a panel session on the subject. The field of artificial intelligence has several objectives:

(1) The development of computational models of intelligent behaviour- both cognitive and perceptual.

(2) The engineering-oriented goal of developing programs that can solve problems normally thought to require human intelligence.

(3) The development of tools and techniques needed for the above two items.

The development of a system intended to meet the needs of users and is intended to provide expert advice falls into the second category.

The field of expert systems is relatively new. It extends back approximately twenty years, although it is relatively recent that such systems have been referred to as expert systems. Although there has, in the past few years, been a great deal of work on this subject, the actual accomplishments have, at best, been modest. In using the term modest, it is meant that with respect to having expert programs used by individuals in their daily work, there are relatively few such systems. In the following section we briefly note some of

the expert systems that have been developed, and their status. In the last section we discuss several issues that must be addressed if expert systems are to become a reality. These issues are posed for discussion, and no positions are taken on them. The intent of the panel discussion is to explore the issues in great depth.

2. Background in Expert Systems

Early work in artificial intelligence was oriented towards providing general approaches to probelm solving. It was realized that some of the problems being attacked were, perhaps, more difficult than anticipated. This was particularly true with work in machine translation and natural language processing. Efforts to apply theorem proving techniques to arbitrary problems in diverse domains introduced combinatorial explosions. A move was therefore made towards specializing problems and building into application areas special knowledge focused on the domain of application. Systems that focus on specific problem domains, building in knowledge specific to that domain, have come to be called expert systems.

There have been several phases in the development of Expert Systems. This may be illustrated by efforts leading to one successful system, MACSYMA, whose function is to act as an expert in the area of formal integration of functions. It is perhaps of interest to note that throughout the development of MACSYMA the term "expert system" was never applied. The first stage was the demonstration that it was possible to perform symbolic integration on a computer. Jim Slagle's system, SAINT, was developed and was subsequently tested. It succeeded in passing an examination in integral calculus at MIT. Although it was successful, it was intended to be a research program, and not a finished product that could be used by scientists and engineers. Following SAINT,

Joel Moses developed SIN, which was able to perform integration in a more powerful way than the Slagle program. It had more general rules built into it, and more explicit answers to problems whose integrals were already known. In the third stage, the system, MACSYMA, was developed to meet the day- to-day needs of scientists and engineers. Thus, after a long research and development stage, a final product was developed. Although MACSYMA is a successful system, in the sense that it is currently in use by scientists and engineers, many individuals fail to refer to it as an expert system. Neither MACSYMA, SAINT nor SIN are referred to as expert systems since the term was not in vogue when they were developed. It is clear, however, that all three systems would be referred to as expert systems were they developed today.

There are several stages in the engineering of an expert system:

Phase 1 - Research in which the feasibility of developing an expert system in a specific domain is established.

Phase 2 - Development of and experimentation with a prototype system.

Phase 3 - Field test the prototype system.

Phase 4 - Use of the expert system in the field.

In discussing the status of a particular "expert system", it is useful to distinguish its stage of development. There are four expert systems that are routinely in use: MACSYMA; DENDRAL(Feigenbaum et al. [1971]), an expert system that analyzes mass spectral patterns to determine the chemical structure of unknown compounds; R1 (McDermott [1981]), an expert system to determine computer layouts and configurations; and PUFF (Osborn et al.[1979]), an expert system that interprets pulmonary function tests.

A list of some representative expert systems and their domains of application appears in Table 1. A number of useful articles on expert systems appear in books (Michie [1979], Hayes-Roth et al.[1983], Webber et al. [1981], Szolovits et al.[1982]). Several comprehensive surveys have been written on expert systems (Duda et al.[1983], Buchanan [1982], Nau [1983]). See Reggia [1982] for a comprehensive list of references in expert systems oriented primarily towards medical applications.

Expert System	Domain	Reference
AQ11	Diagnosis of plant disease	Chilausky et al.[1976]
CASNET	Glaucoma assessment and therapy	Weiss et al.[1978]
DENDRAL	Mass spectroscopy interpretation	Feigenbaum et al.[1971]
Digitalis Advisor	Digitalis dosing advice	Gorry et al.[1978]
Dipmeter Advisor	Oil exploration	Davis et al.[1981]
El	Analyzing electrical circuits	Stallman et al.[1977]
Internist-I	Internal medicine diagnosis	Miller et al.[1982]
HASP & SIAP	Ocean Surveillence (signal processing)	Nii et al.[1982]
KMS	Medical consulting	Reggia [1980]
MACSYMA	Mathematical formula manipulation	Moses [1971]
MDX	Medical consulting	Chandrasekaran et al. [1979].
Microprocessor EXPERT	Protein electrophoresis interpretation	Weiss et al[1981]
MOLGEN	Planning DNA experiments	Martin et al.[1977]
MYCIN	Antimicrobial therapy	Davis et al.[1977]
PROSPECTOR	Geological mineral exploration	Hart et al.[1978]
PUFF	Pulmonary function test interpretation	Osborn et al.[1979]
R1	Computer layout and configuration	McDermott et al.[1981]

TABLE 1 - Representative Expert System

Expert systems have been implemented using a variety of different approaches:

(1) Embedding control and inference in a program written in a language such as FORTRAN or PASCAL (Bleich [1972]).

(2) statistical pattern classification techniques as the basis of making conclusions. For example, Bayseian (Ben-Bassat [1980]), and linear discriminant function (Faught et al. [1979]), have been proposed.

(3) Developing cognitive models of diagnostic reasoning Reggia [1981].

(4) Production rule based systems (Davis et al. [1977]).

3. ISSUES

There are a wide range of issues that have to be addressed before expert systems can reach full maturity. These range from the philosophical to the moral to research issues and to user acceptance. It is not intended that all of the items need be addressed before such systems can become a reality, but that a number of these issues must be developed before the field can reach maturity.

1. Philosophical Issues

   a. What is meant by knowledge and how does one differentiate between data and knowledge?

   b. Will it ever be possible to capture all knowledge in a domain of real interest?

   c. Can one deal with systems in which there are significant gaps in knowledge, and how can one assess the effectiveness of such systems?

   d. How does one differentiate an expert system from an application program?

   e. Can an expert system exhibit intelligence in the same sense as attributed to humans?

2. Moral and Sociologic Issues

   a. Are there classes of expert systems that should never be attempted: they are morally repugnant ?

   b. What are the legal problems? Who is responsible for adverse reactions when a medical expert system incorrectly diagnoses a patient?

   c. What are the potential social consequences of expert systems and

are they for the good, or wil they lead to major social problems?

d. Who should build expert systems: domain experts, computer scientists, or both?

3. Research Issues

Knowledge Acquisition and Representation

a. How does one identify and encode knowledge?

b. What characteristics should a knowledge representation formalism have?

c. How does one express temporal knowledge and physiological mechanisms involved in the evolution of disease processes?

d. How does one represent exceptions to situations?

e. How does one explain the basis for the decision criteria and/or rules used in a knowledge based system?

f. Should knowledge be augmented by using causal and mechanistic links that represent functional behavior?

g. How does one obtain large, reliable data/knowledge bases?

Inference and Uncertainty

a. How does one deal with vagueness and ignorance? Are fuzzy logic (Zadeh [1978]) and statistical theories of evidence (Shafer [1976]) useful?

b. In what ways is logical inference useful?

c. Will indefinite data(i.e., data of the form p V q) be needed for expert systems? What are the implications with respect to the development of such a system or answers obtained during its use?

d. How can logical inference handle exceptions?

e. How is reasoning performed in the presence of ignorance and how can

a reasoning system recognize the limits of its knowledge?

f. What is "common sense" knowledge, and how can it be embedded in expert systems?

Control

a. How is search controlled in an expert system?

b. What is needed to permit the user to exercise control and to understand what the expert system is doing?

c. Do current languages allow for control needed to find solutions to problems in an efficient manner?

Explanation

a. Explanation in terms of goals and its knowledge base is very useful. However, experts who provided a set of rules are likely to give explanations in terms of physiological mechanisms or disease processes. How can a system accomplish this?

b. How does one provide explanations to different classes of users? That is, how does one maintain models of users and provide explanations to the various users according to the implied intent of the user?

c. How can the user be aware of the significance of questions asked by an expert system? (e.g. the expert system may ask if a spinal tap has been performed, and the user should be able to understand why the question is being asked, as well as the fact that this test is potentially dangerous).

4. System Assessment and User Acceptance

a. How does one certify an expert system?

b. How does one assess system performance, particularly where the "correct" solution to the problems may not always be known? (e.g. medical diagnosis)

b. How does one obtain large, reliable databases?

c. How does one scale up a system from small experimental systems? What are the problems?

d. How does one develop user friendly systems?

e. How does one develop systems that can be transferred from the experimental laboratory to a remote user site?

f. When will cost-effective systems be developed?

g. How can user resistance to change be overcome?

h. How will new knowledge and changes be made at the user sites?

We have set forth some of the issues associated with developing expert systems. In the course of the panel discussion we will consider these issues.

5. REFERENCES

1. M. Ben-Bassat, et al. "Pattern-Based Interactive Diagnosis of Multiple Disorders-The MEDAS System", IEEE Trans. Pat. Anal. Machine Intelligence, 2, 1980, 148-160.

2. H. Bleich, "Computer-Based Consultation", AM. J. Med., 53, 1972, 285-291.

3. B. G. Buchanan, in Machine Intelligence, Vol 10, J. E. Hayes, D. Michie and Y.-H. Pao Eds. Wiley, New York, 1982, pp. 269-300.

4. B. Chandrasekaran et al., "An Approach to Medical Diagnosis Based on Conceptual Structures," Proc. Sixth Int'l Joint Conf. Artificial Intelligence, 1979, pp. 134-142.

5. R. Chilausky, B. Jacobsen, and R.S. Michalski, "An Application of Variable- Valued Logic to Inductive Learning of Plant Disease Diagnostic Rules", Proceedings Sixth Annual International Symp. Multiple-Valued Logic, 1976.

6. R. Davis et al., "The Dipmeter Advisor: Interpretation of Geological Signals," Proc. Seventh Int'l Joint Conf. Artificial Intelligence, Aug. 1981.

7. R. Davis, B. Buchanan, and E. Shortliffe, "Production Rules as a Representation for a Knowledge-Based Consultation Program," Artificial Intelligence, Vol. 8, No. 1, 1977, pp. 15-45.

8. R. O. Duda and E. H. Shortliffe, "Expert Systems Research", Science, Vol. 220, No. 4594, 15 Apr 1983, pp. 261-268.

9. E. Faught et al., "Cerebral complications of Angiography for Transient Ischemia and Stroke-Prediction of Risk", Neurology, 29, 1979, 4-15.

10. 10. E. Feigenbaum, G. Buchanan, and J. Lederburg, "Generality and Problem Solving: A Case Study Using the DENDRAL Program," Machine Intelligence 6, D. Meltzer and D. Michie, eds., Edinburgh University Press, 1971, pp. 165- 190.

11. G. Gorry et al., "Capturing Clinical Expertize-A Computer Program that considers Clinical responses to Digitalis", Amer. J. of med., 64,1978,452-460.

12. P.E. Hart, R.O. Duda, and M.T. Einaudi, A Computer-Based Consultation System for Mineral Exploration, Technical Report, SRI International, Menlo Park, California, 1978.

13. F. Hayes-Roth, D. Waterman, and D. Lenet, Eds. Building Expert Systems, Addison-Wesley, New York, 1983.

14. N. Martin et al., "Knowledge-Base Management for Experiment Planning in Molecular Genetics," Proc. Fifth Int'l Joint Conf. Artificial Intelligence, 1977, pp. 882-887.

15. J. McDermott and B. Steele, "Extending a Knowledge Based System to Deal with Ad Hoc Constraints," Proc. Seventh Int'l Joint Conf. Artificial Intelligence, 1981, pp. 824-828.

16. D. Michie, Ed, Expert Systems in the Microelectronic Age, Edinburgh Univ. Press, Edinburgh, 1975.

17. R. Miller et al., New England Journal of Medicine, 307,468(1982).

18. J. Moses, "Symbolic Integration: The Stormy Decade," Comm. ACM, Vol. 14, No. 8, 1971, pp. 548-560.

19. D. S. Nau, "Expert Computer Systems", Computer, 16, 63(1983), pp. 63-85.

20. H. Nii et al., AI Magazine 3,23(1982).

21. J. Osborn et al., "Managing the Data from Respiratory Measurements," Medical Instrumentation, Vol. 13, No.6, Nov. 1979.

22. H.E. Pople, "The Formation of Composite Hypotheses in Diagnostic Problem Solving: An Exercise in Synthetic Reasoning," Proc. Fifth Int'l Joint Conf. Artificial Intelligence, 1977, pp. 1030-1037.

23. J. A. Reggia, "Computer-Assisted Medical Decision Making", Applications of Computers in Medicine, M. Schwartz, Ed, IEEE, 1982.

24. J. A. Reggia and B. Perricone, "Knowledge-Based Decision Support Systems - Development Through High- Level Languages", Proc. Twentieth Ann.

Technical Symposium of Washington DC, ACM, June 1981, pp. 75-81.

25. J. Reggia et al., "Towards an Intelligent Textbook of Neurology," Proc. Fourth Annual Symp. Computer Applications in Medical Care, 1980, pp. 942-947

26. G. Shafer, A Mathematical theory of Evidence, Princeton Univ. Press. Princeton, New Jersy, 1976.

27. R.M. Stallman and G.J. Sussman, "Forward Reasoning and Dependency-Directed Backtracking in a System for Computer-Aided Circuit Analysis," Vol. 9, Artificial Intelligence, 1977, pp. 135-196.

28. P. Szolovits, Ed, Artificial Intelligence in Medicine, West View, Boulder, Colorado, 1982.

29. S.M. Weiss et al., "A Model-Based Method for Computer-Aided Medical Decision- Making," Artificial Intelligence, Vol. 11, No. 2, 1978, pp. 145-172.

30. S. Weiss and C. Kulikowski, "EXPERT - A System for Developing Consultation Models," Proc. Sixth IJCAI, 1979, pp. 25-40.

31. S. M. Weiss et al., In Proc. of the Seventh Int'l. Joint Conf. on AI, Univ. of British Columbia, Vancuvar, Aug 1981, pp. 835.

32. L. Zadeh, "Fuzzy Sets", Systems 1,3(1978).

KNOWLEDGE REPRESENTATION IN AN EFFICIENT DEDUCTIVE INFERENCE SYSTEM

E. P. Stabler, Jr.
E. W. Elcock

University of Western Ontario
London, Canada

ABSTRACT

Efficient response to queries addressed to a large data base is an important problem of knowledge representation. The problem and various solutions have been well researched for certain "conventional" (e.g. relational) data models. The analogous problem has been tackled and solutions similar in spirit to those for relational models developed for data bases and queries expressed in Horn clause systems such as Prolog with the severe constraint that the "data base" is a set of ground instances of assertions.

The situation becomes more interesting and challenging when the data base is deductive: e.g. a Prolog first-order theory. Basically the interest is in finding an automatic way of representing the first-order theory which facilitates the dynamic reordering of residual literals and the selection of the next goal to be evaluated based on a changing measure of the cost of evaluating each goal in the residual query.

The current paper presents partial solutions which can be used to obtain dramatic reductions in search times. The paper also identifies some remaining and difficult problems.

The methodology was designed with the processing of natural language queries in mind, but it is quite general in its domain of application.

KNOWLEDGE REPRESENTATION IN AN EFFICIENT DEDUCTIVE INFERENCE SYSTEM

E. P. Stabler, Jr.
E. W. Elcock

University of Western Ontario
London, Canada

There is an increasing demand for large database systems that provide efficient inference capabilities. These are obviously needed in question answering systems and expert systems, but their potential range of application is really very wide. It is an advantage to allow any database user a uniform view of both explicit and implicitly represented information. Accessing the database through a deductive inference system offers the possibility of such freedom, first, in its ability to decide whether any particular proposition follows from what is represented in the database, and second, in its ability to use rules, meaning postulates and definitions of new terms and relations as nonlogical axioms in making such decisions. Thus, rather than having to search for particular pieces of information, the user can simply ask whether or not a particular proposition follows from the database: general rules can be introduced to cover large classes of particular facts and to define new terms that might be used in queries.

Keeping the deductive access to a large database efficient requires, in the first place, that we deal with some of the standard database management problems, viz., the problems of making search efficient and of optimizing queries to minimize the need for search. These problems are paricularly pressing when the database and access system are to be embedded in a larger design, such as a question answering system. In this context the system must interface with natural language processors, rather than with the typical brilliant and insightful human database user who can learn how to avoid the system's weak spots. As a result, database queries cannot be expected to arrive in a form that is optimal from the point of view of efficiency. In this report we will show how such standard database management problems can be handled in PROLOG, one of the most efficient and most widely known theorem proving systems.

---

* The software described in this paper was designed and implemented by a research group which included the authors, D. Wyatt, and A. Young. This work is also described in Elcock et al.(forthcoming) and Stabler (1982).

PROLOG preliminaries. Since PROLOG is fairly well known and there are good introductions to the language (e.g., Clocksin and Mellish, 1981) we will only briefly review some important features. PROLOG is basically a Horn clause theorem prover. It also has metalogical facilities that can provide higher order effects, and of course nonstandard effects can be obtained by quantifying over possible worlds (cf. Moore, 1980). A clause is a first order prenex formula whose prefix consists of (only) universal quantifiers and whose matrix is a disjunction of literals, where a literal is an atomic formula or the negation of an atomic formula. Since the order of universal quantifiers makes no difference, they do not need to be written down. A Horn clause is a clause whose matrix contains at most one positive literal. The restriction to Horn clauses does not, in principle, prevent us from expressing anything that can be expressed in first order logic. The relevant results are these: for any formula F of first order predicate calculus, there is an easily constructible set S of clauses which is inconsistent if, and only if, F is (see e.g., Chang and Lee thm.4.1.); and for any set S of clauses there is an easily constructible set H of sets of Horn clauses such that there is an inconsistent set in H if, and only if, S is inconsistent (see e.g., Henschen and Wos, 1974). In spite of this generality in principle, though, some problems are much more feasible and natural when expressed in non-Horn clause form.

The restriction to the Horn clause subset of first order logic is not the only special logical problem that faces a PROLOG database system. In the first place, we should note that PROLOG does not immediately provide ideal inference capabilities even within the Horn clause logic: it's failings are the familiar ones. It is well known that "Horn sets" are not decidable (Hermes, 1965), and so of course PROLOG cannot decide whether an arbitrary query of its Horn clause logic follows from the database or not, even given unlimited time and space. And although it is easy to design proof methods for Horn clause logic which are complete in the sense that they will find a proof of an arbitrary sentence if there is one, the most efficient theorem provers, like PROLOG, are not complete. They are even unsound in the sense that they will sometimes claim to have found a proof when there is no valid proof. Let's consider these problems briefly before considering the more standard database problems.

Soundness. It is well known that PROLOG will sometimes produce an invalid proof. For example, given the database "p(X,X).", PROLOG will say that the query "p(Y,f(Y))." follows. The instance that follows, according to the PROLOG system, is the one in which Y=f(f(f...(f(Y)))).... (Since this is an infinite expression, there will be trouble if PROLOG tries to print it out.) But obviously,

"(Ey)(P(y,f(y)))" does not follow from "(x)(P(x,x))". PROLOG gets this incorrect result because it does not do the "occurs check" in the course of unification. What it does is this. When confronted with the query "p(Y,f(Y))" it tries to match it with the database clause "p(X,X)". It begins with the first argument; in effect, "Y" is substituted for "X". The result is "p(Y,Y).", and this is identical to the query up to the second argument. So now PROLOG tries to match the second arguments, which it does by substituting "f(Y)" for "Y". As a result, Y=f(f(f...(f(Y)))...., and the query is judged to be an instance of the database clause. Strictly speaking, this matching process is not the unification which is employed in sound resolution procedures, because a variable cannot properly be unified with any term which contains that variable. Implementing unification correctly would involve performing an "occurs check" to make sure that the variable does not occur in the term it is being unified with. Performing this check in every unification step is expensive, especially when the terms being unified are large. Since the matching process without an occurs check is so much cheaper, and since it is sufficient in most cases, most PROLOG implementations do not use strict unification. As noted above, assuming that we do not want to just tolerate errors, this means that the users of these systems must make sure they are not accepting conclusions based on unsound inferences. There are a number of ways to do this.

One way to avoid unsound inferences is simply to require that only ground clauses occur in the database. This restriction is perfectly straightforward, and it is obviously adequate since the database would then not contain any variables which might occur in terms they would be matched with. The problem is that this restriction is obviously going to eliminate the features which make logic programming languages particularly attractive. (Consider, for example, the features mentioned at the beginning of this paper.)

There are other similar and less restrictive strategies. We can hope that programmers experienced with PROLOG will learn how to avoid creating a database in which unsound inferences will be made. If an unsound inference is going to cause trouble, they should block it. The problem then is not with the "system" database which is provided by programmers, but with database clauses which might be provided by naive users. We do not want to require the users to understand and attend to such things as the peculiarities of the PROLOG matching process. So we can either provide a complete system to which the user cannot add information, or we can require that any information added be ground clauses. The "system" database would then be created by programmers familiar with PROLOG's matching process, and the "user" database, if there is one, would not add any dangers of unsoundness. This restriction on the users' database would certainly to be felt,

however; it severely constrains what the user can do with the system. He would not, for example, be able to add definitions of new relations in terms of relations already provided by the system. The only other alternative that seems to be available, though, would be to do the occurs check whenever non-ground clauses that could conceivably cause an error are used. Since the main thrust of the present project is to allow the database users the real advantages of accessing a database through an inference system (without errors, even very unlikely ones!), this last strategy is the only acceptable one, and is currently being explored.

Completeness. The second problem that we would like our system to deal with as well as possible is that of avoiding attempts to find proofs which are beyond the theoretical capabilities of PROLOG. We have already noted the obvious point that mere completeness is not going to do us any good if the proof procedure is just not feasible. But the point of interest is that if finding a proof of some result is beyond PROLOG's theoretical capabilities, it is of course also beyond its practical capabilities. It is a good strategy to try to keep the whole class of proofs that might be sought within the theoretical capabilities of PROLOG, and then to keep those proofs as efficient as possible. Sometimes a simple change in the database, query, or proof strategy that brings a result within the theoretical capabilities of the system also suffices to bring the result within the practical capabilities of the system.

The following familiar sort of example illustrates this situation. (This example is taken from Moore(forthcoming), where it is used to illustrate the related problem of forward vs. backward chaining.) One of the standard ways to define a relation is with "base rules" and "induction rules." For example, the one-place relation or property of being Jewish might be partially defined with a list of people who are Jewish and with the rule from the Talmud that a person is Jewish if the mother of that person is Jewish, as follows:

```
jewish(bar-hillel).
jewish(X):-jewish(mother(X)).
```

Given this database, PROLOG will properly indicate that there is a proof of the query "jewish(bar-hillel).". If, however, the clauses in the database are reversed, putting the "induction rule" before the "base rule," PROLOG will never succeed in finding a proof of this query. Because it uses a depth-first proof strategy and selects the first database clause first, it would never get to the second rule, the base rule, which it would need to use. It would "loop," using the first rule, the "induction rule," over and over again. Since this sort of situation is quite common, we can adopt the

strategy of always putting "base" rules before "induction" rules. The problem is to recognize them. A crude approximation that will handle this case is to check each clause that is going into the database to see if it is a simple assertion, a unit clause with an empty body. If it is, put it at the beginning of the list of clauses which have the same predicate; otherwise, put it at the end of the list. (This is one of the things which is done by our predicate "update" which will be described in more detail later.) It should be noted, however, that this ordering strategy will not work in cases where the "base" rules are not simple assertions, and it will not work in cases in which there is more than one "induction rule." There are cases of incompleteness which cannot be removed by any reordering of database clauses. (Cf. Elcock, 1982; 1983.) Thus, our implementation of this ordering strategy is not motivated so much by completeness considerations as by feasibility: it is generally cheaper to find solutions using unit clauses, so these should be considered first.

Feasibility. Problems which are at present effectively insurmountable also seem to face the general goal of staying within the practical limits of the system. The use of a language that has a formal, logical interpretation is no panacea for the standard sorts of programming problems; we do not have any mechanical method for transforming logically correct but inefficient code into correct and efficient code. The ordering method just described will help in some cases. Another thing that is done (by "update") to improve efficiency is that whenever a clause is added to the database, all instances of that clause are deleted. So, for example, the addition of "p(X)." to the database will cause "p(a)." to be deleted. And the addition of "p(X,Y)." will cause "p(X,X)." to be deleted. So a certain easy to find redundancy is automatically eliminated. Apart from such simple steps as these, though, there is not much that can be done cheaply and easily to enlarge the class of feasible proofs except to provide as much time and space as is practical, to minimize the need for unnecessarily long searches, and to make searches of the database as efficient as possible. Search efficiency can be improved by indexing the database; unnecessary search can be eliminated with appropriate goal selection strategies and intelligent backtracking. Each of these methods will now be considered in turn. Notice that none of them are theorem proving matters; they are metalogical operations that change the set of axioms from which we may draw inferences. They can be taken care of automatically, out of the sight of the user. The user should see only the improved efficiency.

Indexing the database. A standard technique for making search efficient involves indexing the units of information so

that when an item is needed the whole memory does not need to be searched; instead the location of the needed information can be looked up in an array or hash table. The DEC-10 PROLOG interpreter indexes database clauses according to their "head" predicates, i.e., according to the predicate in the head of each clause (Pereira et al., 1978). But when a relation is large, when there are many clauses with a particular head predicate, the searches will still be long. In this situation, the standard strategy is to start secondary indexing on the arguments of the relations. A database that is indexed for every argument of every relation is said to be totally indexed or totally inverted. Some PROLOG implementations, such as IC-PROLOG, provide facilities for indexing according to the principal functor of arguments to the head predicates in the database (Clark and McCabe, 1982). And in systems like interpreted DEC-10 PROLOG, secondary indexing effects can be obtained simply by building auxiliary predicates which incorporate names of the principal functors of the arguments. This technique was used in the Edinburgh Chat-80 system (Warren, 1981; Warren and Pereira, 1981), and we used it in our work.

Goal selection strategies. The order in which the goals of a query are solved can make a substantial difference in resource use. Suppose, for example, that the database has 4000 clauses with the predicate "g1" and 1 clause with the predicate "g2", and that all of these are ground clauses. Then, given the left-to-right selection method that is standard in PROLOG, and assuming that the database is totally indexed, it is much more efficient to evaluate the query,

```
g2(X,Y),g1(X,Y).
```

than it is to evaluate the query,

```
g1(X,Y),g2(X,Y).
```

Evaluating the latter query could involve an enormous amount of backtracking. Evaluating "g2(X,Y)" first, on the other hand, immediately provides the only instances of "X" and "Y" which could possibly satisfy the query. The indexing will allow this instance to be checked without a long search, and, in any case, backtracking is more expensive than a simple search for a matching head predicate. So in general, we want to evaluate the least expensive goals first. When the database is all ground clauses and the query has variables in all argument places, we can let the cost of a goal be the size of the relation, i.e., the number of clauses in the database whose heads have the same predicate as the goal. The cost function should be more elaborate, however, when the database contains clauses with variables (or terms containing variables) or the query contains goals with non-variables.

Let's consider first the elaboration of the cost function which is needed to allow for queries with instantiated arguments. If a predicate is indexed in the database, then

any "user" query of that predicate will be solved by first converting it into its indexed form and then finding a solution to that "indexed" query. In a totally indexed database the cost of solving the original query will not in general depend on the size of its main predicate, but rather on the size of the sets of arguments that occur in each of the n-positions of any n-place predicate in the query, since these are the arguments to the indexed predicates. In order to estimate the expense of finding a solution to a query (in a manner which will be described below) we can keep records of the sizes of the sets of arguments that occur in each place of every predicate. When all the database clauses are ground clauses, calculating the sizes of these sets is straightforward. The sets simply include all the different terms that occur in the relevant argument positions.

This brings us to the question of how to elaborate the cost function to make it appropriate for a totally indexed database that is not restricted to ground clauses. In this situation, not all of the possible instantiations of any particular argument position need be explicitly available; some of them will only be found by the inference process. We do not want to have to calculate all of the possible instantiations of each predicate, so we need some reasonable way of estimating the number of distinct terms that could occur in each argument position. The details of the calculation will not be described here, but roughly, we make worst-case assumptions that allow us to calculate the maximum number of possible distinct provable instantiations of each predicate. And then, thinking of each different predicate as a relation, we want some reasonable way of calculating the relation size. Again, we calculate relation sizes by making a worst-case estimation of the number of solutions one would be able to find to the query consisting of any particular predicates followed by the appropriate number of variables. We calculate these estimates and revise them when new information is added as part of the "updating" process. Given these estimates, we are able to use the same cost estimation formula as was used in the Chat-80 system for ground clause databases. The cost of solving a goal is defined as the size of the relation divided by the product of the argument domain sizes associated with argument positions that are instantiated at the time a solution is sought.

Notice that, given this definition, the cost of a goal may change when other goals in the query are solved. For example, in solving the query,

```
g1(X,Y),g2(Y,Z),g3(Z,a).
```

the solution of the first goal will instantiate the first argument of the second goal, making it cheaper to solve. And the solution to the second goal will leave no uninstantiated arguments in the third goal. So if we want to plan our queries in such a way that the cheapest goal will always be

the next one solved, we will have to anticipate the instantiation of the relevant variables. This process interacts with the backtracking strategies described below, so let's consider those before describing how this query planning should be done.

Selective backtracking. Sometimes PROLOG will do a lot of unnecessary backtracking in the course of finding the set of solutions to a query. Consider, for example, the query,

```
bagof(X,h(X),B).
```

where the unary predicate "h" is defined by the database clause,

```
h(X):-g1(X),g2(Y).
```

And suppose the database provides some number n of solutions to the first goal, "g1(X)", and some very large number m of solutions to "g2(Y)". In finding the list B of solutions to "h(X)", a solution to the first goal "g1(X)" will be found; then a solution to the second goal "g2(Y)" will be found and the instance of "X" will be put in list B. The system will then backtrack to find all m solutions to the second goal, putting the first solution to the first goal in the list B each time. Since we are only interested in getting the instances of "X" which satisfy the goals given, it is just a waste to get each such solution m times. We could use "setof" instead of "bagof" to get a nonredundant list of solutions, but this query also wastes the time to get all the redundant solutions before deleting them. Interchanging the positions of "g1(X)" and "g2(Y)" does not improve things. And simply putting a cut into the original query somewhere will also not achieve the goal of getting a complete set of the instances of "X" without this wasted effort. (In this case we could interchange the goals and put a cut between them, but this sort of solution will not always be available, as the examples below will illustrate.) Because it shares no variables with the head of the clause, the goal "g2(Y)" is, in effect, an independent subproblem; it must have a solution, but this is all we need to know to find all of the solutions to "h(X)".

Precisely the same situation arises if instead of having a definition of "h", we simply ask,

```
bagof(X,(g1(X),g2(Y)),B).
```

We would like to be able to avoid the unnecessary backtracking in all such cases.

This problem was handled in the Chat-80 system by putting independent subproblems inside braces, and then changing the PROLOG interpreter so that it would evaluate queries containing such braces appropriately. We used the standard interpreter and used new rules with cuts to achieve the same effect. Thus, instead of evaluating a query like

```
bagof(X,(g1(X),g2(Y)),B).
```

or putting a rule in our database like,

```
h(X):-g1(X),g2(Y).
```

we would enter the auxiliary rule,

```
r1(Y):-g2(Y),!.
```

and then evaluate the equivalent query,

```
bagof(X,(g1(X),r1(Y)),B).
```

or put the following equivalent rule into our database,

```
h(X):-g1(X),r1(Y).
```

The latter query and rule will yield the same results but without all the unecessary backtracking and inference. The body of the auxiliary rule is appropriately evaluated as an "independent subproblem." The savings in resource use can obviously be enormous.

Extending this sort of treatment to more complicated queries and rules is not trivial, but not terribly hard either. Consider the following sort of case, for example,

```
h(X,Z):-g1(X),g2(Y),g3(Z).
```

In this case we do not want to enter the auxiliary rule,

```
r1(Y,Z):-g2(Y),g3(Z),!.
```

and change our original rule to,

```
h(X,Z):-g1(X),r1(Y,Z).
```

since this procedure would only allow us to find one of the possibly many solutions to g3(Z). The moral of this sort of case is that no head variable should occur uninstantiated in a subproblem when that subproblem is evaluated. Thus, although "g3(Z)" should not be included in a subproblem in this last example, it could be included in a subproblem in

```
h(X,Z):-g1(X,Z),g2(Y),g3(Z).
```

In this case the mentioned auxiliary rule would be appropriate, since the head variable "Z" will always be instantiated at the time "g3(Z)" is evaluated, and so its occurrence in an independent subproblem will not restrict the number of solutions found.

Another sort of case that can arise is that we may have subproblems within subproblems. Consider for example the query,

```
h(W):-g1(X)-g2(X,Y),g3(X,Z).
```

None of these goals contain head variables, so they can immediately be put into an independent subproblem. After the first of these goals has been solved, though, the remaining two goals do not share any variables, so they break into two further subproblems. Accordingly, the rule would be handled by transforming it into,

```
h(W):-r1(X,Y,Z).
```

and then we enter the following auxiliary rules,

```
r1(X,Y,Z):-g1(X),r2(X,Y),r3(X,Z),!.
r2(X,Y):-g2(X,Y),!.
r3(X,Z):-g3(X,Z),!.
```

The rationale for doing this is just the same as above. Suppose for example, that for some choice of "X" we are unable to prove "g3(X,Z)". There is no point in backtracking to find other solutions to "g2(X,Y)", since the choice of "Y" is irrelevant to our problems with "g3(X,Z)". What we need to do

is immediately go back to find another choice of "X". This is precisely what our new rules will accomplish.

This grouping of goals into subproblems is sometimes going to interact with our goal selection strategy. For example, after ordering the goals on the basis of solution cost, it may turn out that an independent subproblem is broken up by a goal containing a head variable. This sort of conflict is resolved with an optimizing algorithm which integrates the cost planning and the selective backtracking strategies we have described.

Optimizing. The optimizing algorithm that was implemented is roughly the following:

Given a rule of the form H:-G1,G2,...Gn,

(1) Order the list of goals, G1, G2,..., Gn, according to solution cost, as discussed above.

(2) Look through the goals, in order, to find head variables.

(i) If such a a goal is found, it will be the cheapest goal containing a head variable, so move it to the front of the list of goals, and assume for the remainder of the optimizing process that its arguments are instantiated. (Some of them may occur in other goals.) Consider only the remaining goals for the rest of the optimizing process. Reorder these goals according to cost, and repeat step (2).

(ii) If no such goal containing head variables is found, proceed to the next step.

(3) Any goals that remain to be considered at this point will not have any head variables at the time they are to be solved, so they constitute independent subproblems. Take the first goal Gi on the list - it will be the cheapest - and check the following goal to see if it shares any variables with Gi. If it does, it is to be included in the same subproblem with Gi, and check the next goal to see if it contains any of the same variables as Gi, and so on until there are no more goals or until a goal with no variables in common with Gi are found. At this point we have a list of the goals in the Gi subproblem, and possibly also a list of remaining goals not in the Gi subproblem. Now enter an auxiliary rule, "the Gi rule," into the database. The Gi rule is given a unique head predicate and has as head arguments all the variables that occur in the goals of the subproblem. The body of the Gi rule consists of the goals in the Gi subproblem. We now want to optimize the body of this rule as well, so assume for the remainder of the optimizing process that the variables in Gi are all instantiated. Reorder the rest of the goals in the body of Gi rule (if any) and perform this step (3) again on these goals to

find subsubproblems. Finally, reorder the list of goals outside of the Gi subproblem and perform this step (3) on them as well.

This algorithm anticipates the instantiation of variables both in its cost calculations and in its recognition of independent subproblems. It appears to be a very expensive process, but it need only be done once for any rule being put into the database, and it can actually save an enormous amount of time.

Suppose that our database contains one ground clause with the predicate "g1", one hundred ground clauses with the predicate "g2", five hundred ground clauses with the predicate "g3", and nothing else except the following definition of the predicate "h":

```
h(X):-g3(Y),g2(Z,Y),g1(X).
```

Now consider the query,

```
setof(X,h(X),S).
```

This query is obviously maximally inefficient, but our database is not really huge and so it may not be obvious that it would be worth optimizing the rule for "h(X)". The actual processing times are as follows. Executing the maximally inefficient query in the situation described takes 2291 ms. Optimizing the rule for "h" tranforms it into,

```
h(X):-g1(X),r1(Y,Z).
```

and enters the auxiliary rules,

```
r1(Y,Z):-g2(Y,Z),r2(Z),!.
r2(Z):-g3(Z),!.
```

This optimizing process takes about 280 ms. And executing the same "setof" query, but now with the optimized definition of "h" and the auxiliary rules, takes about 30 ms. Obviously, the optimizing is worthwhile in any case like this one. On a larger database, the improvements are even more dramatic, as would be expected. The optimizing code could also be compiled to improve its efficiency further once it has been put in the form in which we want to use it in any particular application.

Conclusion. The work that has been described here is aimed at providing the basis for a feasible, pragmatic deductive inference system. It is completely general and portable. The applications that this work is specifically designed for are those in which a user wants to have interactive deductive access to a database which may include general rules (expressions containing logical variables) as well as particular facts (expressions containing no variables). This sort of application would go substantially beyond most previous logic programming projects which usually require that the database contain only ground clauses or that the user cannot add new rules. It is precisely the more general sort of database system that exploits the real advantages of a deductive system, though, and this sort of system would be required in many question answering systems.

References.

Chang, C. and Lee, R.C. (1973) Symbolic Logic and Mechanical Theorem Proving. New York: Academic Press.

Clark, K. L. and McCabe, F. (1982) IC-PROLOG - language features. In K.L. Clark and S.-A. Tarnlund, eds., Logic Programming. New York: Academic Press.

Clocksin, W.F. and Mellish, C.S. (1981) Programming in PROLOG. Berlin: Springer-Verlag.

Elcock, E.W. (1982) Goal selection strategies in Horn clause programming. Proceedings of the Fourth National Conference of the Canadian Society for Study in Artificial Intelligence.

Elcock, E.W. (1983) The pragmatics of PROLOG - some comments. Unversity of Western Ontario, Department of Computer Science Technical Report.

Elcock, E.W., Stabler, E.P., Wyatt, D., and Young, A. (forthcoming) Database management in PROLOG. Unpublished technical report.

Henschen, L. and Wos, L. (1974) Unit refutations and Horn sets. JACM, 21, pp 590-605.

Hermes, H. (1965) Enumerability, Decidability, Computability. New York: Springer-Verlag.

Moore, R.C. (1980) Reasoning about Knowledge and Action. SRI Technical Note 191.

Moore, R.C. (forthcoming) The role of logic in knowledge representation and commonsense reasoning.

Pereira, L.M.; Pereira, F.C.N. and Warren, D.H.D. (1978) User's Guide to DECsystem-10 PROLOG.

Stabler, E.P. (1982) Database and theorem prover designs for question answering systems. Centre for Cognitive Science technical report, Cogmem No. 12, University of Western Ontario.

Warren, D.H.D. (1981) Efficient processing of interactive relational database queries expressed in logic. Department of Artifical Intelligence Research Paper No. 156, University of Edinburgh.

Warren, D.H.D. and Periera, F.C.N. (1981) An efficient easily adaptable system for interpreting natural language queries. Department of Artificial Intelligence Research Paper No. 155, University of Edinburgh.

A LOGIC-BASED EXPERT SYSTEM FOR MODEL-BUILDING IN REGRESSION ANALYSIS

Ferenc Darvas, Kornél Bein, Zoltán Gabányi
Company for Computer-Assisted Drug Design*,
1054 Budapest, Akadémia u. 17.

1. Introduction

Methods of mathematical statistics and pattern recognition make a significant part of computer applications besides data processing. Although there are numerous expert systems based on these methods /REG81/, relatively few attempts have been made for automatic building of models, computations to be based on, as well as for automatic evaluation of results, which form a major part of brain-work.

This situation seems to be apt also to regression analysis, the most wide-spread method of mathematical statistics which has been referred to in altogether two publications on possible automation.

In this paper application of logic programming in automatic model-building for regression analysis is presented, in connection with a drug design problem. After a survey of the regression problem /Chapter 2/, the drug design problem and the logical model for problem solving will be dealt with /Chapter 3/. Chapter 4 gives a summary of the program system implementing the model, while Chapter 5 reports on experiences with the system and evaluates the chosen logic programming method.

---

* A joint company of the Institute for Coordination of Computer Techniques, Budapest, and the Institute of Enzymology, Biol. Res. Cent., Hungarian Academy of Sciences, Budapest.

2. The regression problem

It is a frequent case that in regression equations of the form

$$\bar{y} = \bar{\bar{x}}\,\bar{b} + \bar{\varepsilon} \qquad /1/$$

some of the $\bar{x}_i$ column vectors of $\bar{\bar{x}}$ matrix are binary vectors, i.e. the components of x_i can take the discrete values of 0 or 1 only /DRA66/. Components of such vectors can be regarded as logical variables of values "true" or "false" and can be subjected to Boole /AND, OR, NOT/ operations.

Suppose that a variable x_i is assigned to the column vector $\bar{x}_i$, and values of 1 and 0 of x_i correspond to the presence and absence of β_i.

If the model permits to interpret physically a common variable for β_i and β_j, a new vector $\bar{\eta}$ can be introduced, as the logical OR of x_i and x_j:

$$\bar{\eta} = \bar{x}_i \quad \text{OR} \quad \bar{x}_j = \bar{x}_i + \bar{x}_j \qquad /2/$$

More generally

$$\bar{\eta} = \sum_{k=1}^{l} \bar{x}_k \qquad /3/$$

Similarly, a new vector can be generated by performing the Boolean AND operation on original vectors x_k:

$$\bar{\tau} = \prod_{k=1}^{l} \bar{x}_k \qquad /4/$$

Logical combinations according to /3/ and /4/ might be useful in all cases when x_i-s are not competely independent of each other /as it is the common case in many fields/. Causal interpretation of the regression equation requires, however, that each variable,

formed by Boolean operation, should have a meaningful interpretation in the frame of the model investigated.

3. Prediction of drug activity

Drug design aims at predicting biological activity of not yet synthetized or, at least, not yet tested compounds /HAN69/. In its most widely applied approaches /HAN63, FRE 64/, linear relationships between two groups of quantitative descriptors are searched. The first group relates to the biological activity, the second one to the chemical structure of a series of organic compounds. An important group of chemical descriptors is formed by the so-called "indicator" variables, giving the presence or absence of definite groups within the molecules /MAR78/. In drug design methods like in the Free-Wilson approach, Fujita-Ban approach and Kubinyi's mixed method, eq. 1. comprises exclusively or additionally indicator variables as $\bar{x}_i$ column vectors of $\bar{\bar{x}}$. /FRE64, FUJ71, KUB763/.

Indicator variables in eq. 1. can be interpreted as logical variables and also combined in sense of eq. /3/ or /4/ /GOL80/. Because of the high number of the possible logical combinations, the regression eq. 1. cannot be solved with a pre-determined set of all combined variables. Input variable set of the normally used stepwise regression program might include only those logical combinations, which can be interpreted in the context of the biological activity investigated.

Interpretation of the large quantity /sometimes several hundred/ of combined variables means a formidable work, which is burdened with errors of subjective decisions. We think that the high intellectual expenditures of such interpretations compose the main reason for the fact that logical combinations are rarely used in drug design calculations /HAN75, ELG82/, though the first

examples were published a long time ago /BOC65, KOP65, SCHA75, KUB762/.

In order to find physical interpretation for the combined variables by logical programming, a logical model of the drug-receptor "reaction", the ultimate scene of the drug action is needed. Here we give only an informal summary of the most important concepts we have used in our model.

It is supposed that all compounds act with the same "reaction mechanism" on the same macromolecule /receptor/ within a living cell. Measured biological activity values used in eq. 1. are originated almost exclusively from this reaction.

Structure of the compound series can be described as aggregate of unique groups /occuring only in some compounds/ and of the remaining part of the molecule /supposed to be common in all compounds/. Indicator variables of eq. 1. denote the presence or absence of single groups in each compound.

Contributions of the i-th and the j-th indicator variables / γ_i and γ_j/ to the biological activity of all compounds are expressed as the regression coefficients b_i and b_j. b_i and b_j depend, in the first place, on the sets of chemical and physical properties /ε_i and ε_j/ of the chemical groups β_i, and β_j.

There are two cases:
1. Contributions of γ_i and γ_j are independent from each other.
2. Contributions of γ_i and γ_j depend on each other.

In this latter case, it is supposed that γ_i and γ_j enter into an "interaction". Such interaction can be originated e. g. from the formation of an intramolecular chemical bond between the

substituents /FUJ71/. An interaction between γ_i and γ_j depends on the "environmental" conditions in addition to the properties of ε_i and ε_j. Most important groups of the environmental conditions are the

- electronic connections between two groups, transmitted through the common part of all molecules, and
- through-space connections between two groups being able to reach each other.

Let describe δ_{ij} and μ_{ij} the two sets of connections between γ_i and γ_j.

The sets $\varepsilon_1, \varepsilon_2, \ldots \varepsilon_Q$ are constructed in such manner, that they give a possible full description of the meanings of $\gamma_1, \gamma_2 \ldots \gamma_Q$ groups, attaching all to the same position (P_1) of the common part of molecules.

A variable resulted from logical addition of $\gamma_1, \gamma_2, \ldots \gamma_N$, where $N<Q$ according to eq. 3. can be interpreted as the largest common subset within $\varepsilon_1, \varepsilon_2, \ldots \varepsilon_N$ which, in the same time, does not occur in the $\varepsilon_{N+1}, \varepsilon_{N+2} \ldots \varepsilon_Q$ property sets of the remaining $\gamma_{N+1}, \gamma_{N+2}, \ldots \gamma_Q$ groups attaching to P_1.

Logical multiplication of γ_1 and γ_2 according to eq. 4. can be interpreted as an indication to an interaction, I_1.
I_1 occurs if properties of γ_i and γ_j, $\varepsilon_i' \in \varepsilon_i$ and $\varepsilon_j' \in \varepsilon_j$ enable chemical or physical "modification" of β_i and β_j trough the connections $\mu_{ij}' \in \mu_{ij}$ and $\delta_{ij}' \in \delta_{ij}$.

Thus, they can be represented in form of a production rule /SHO76/

$$\varepsilon_i', \varepsilon_j', \delta_{ij}', \mu_{ij}' \longrightarrow I_1 \qquad /5/$$

Subsets of properties can cause several interactions. On the other hand, an interaction can be triggered by several combination of property sets.

4. Computer implementation of the model

Biological activities and chemical structural descriptors of the tested compound are the starting data of the calculations. Structural descriptors can be divided into two classes: binary indicator variables and continuous physico-chemical, quantum-chemical variables. Values of these continuous variables can be calculated by other programs or retrieved from the data base stored in the system. They can be transformed in different ways /addition, subtraction, taking logarithm/. Structural formulae of the compounds with activities to be predicted also belong to the starting data set.
First step of the program is to solve a so-called Fujita-Ban equation system /FUJ71/ using the measured activities and the indicator variables. The solution serves as input for the logical interpretation of the possible logical additions.

In the PROLOG program providing the interpretation, each chemical group /"substituent"/ is characterized by a chemical property set. If a contraction /i.e. logical addition/ is carried out, common part of the property sets of the groups is generated. A contraction is permitted only if the property set of the other chemical groups at the same substitution site do not imply this common part. As interpretation of the contracted variables the resulted common property sets are considered.
After the evaluation of the possible interpretations the computation goes on in an interactive way so that the user decides on the contractions of the substituents step by step. After the user's giving two or more substituents to be contracted, the system computes the new regression equation, its statistics and optionally the estimated biological activities of the untested compounds. If the user accepts the equation, the next contraction will be carried out on the base of the new indicator variable set, otherwise the processing continues

using the previous variable set. Contractions of indicator variables are carried out at each substitution site, one by one.

Having performed all necessary contractions, generation of interaction variables follows. An interaction variable corresponds to the common presence of substituent M at site I and substituent N at site J /logical 'AND' relation/.
Value of this variable is 1 for a given compound if this 'and' relation is true, otherwise it is 0.The system generates all interaction variables with value = 1 for two or more tested compounds, and calculates frequencies for them.

The user gives a lower frequency limit f_1. This specifies that henceforth interaction variables of frequency greater than or equal to f_1 are treated only.

The interaction variables are then prescreened: the program lists the variables with statistics informing about their importance. If a variable is redundant in statistical point of view, it is ignored. The accepted interaction variables are added to the set of variables, and they are also passed to the program giving automatic interpretation.

Using the built-in automatic deductive mechanism, this program tries to prove the interaction rules stored in its data base. If the proving procedure is successful, the user is informed about the result as a possible interaction. The interpreted interaction variable can be included in the input data set for the calculation of the final regression equation. This input involves not only the original, contracted and interaction variables but the continuous ones as well. There are two means for computing the final equation: interactive or automatic stepwise regression analysis. As result of the numerical calculations one gets the regression equation corresponding to the wanted quantitative structure-activity relationship,

its statistical characteristics and the estimated activities of the tested and untested compounds.

5. Experiences. Summary

The system is implemented on the Siemens 7536 computer of the Institute for Coordination of Computer Techniques, in FORTRAN and MPROLOG. MPROLOG is a modular version of PROLOG, being developed by the Institute. Besides its comfortable program development facilities it permits modular structuring of the program.

In order to test drug design performance of the system, earlier results were recalculated and new problems were solved. Recalculating one of our earlier series of structure-activity regression equations /DAR80/ resulted significant and meaningful equations in all of the 8 cases investigated. In addition, an earlier version of the system helped in a great extent to calculate quantitative structure-activity relationships for antifungal nitroalcohols /LOP83/. In summary, the mechanical interpretation of the combined variables seems to be a helpful tool in model-building for drug design.

On the other hand, drug design has been a favourite field of logical programming for a long time. Besides a system for predicting drug interaction /DAR75, FUT76, FUT771, DAR78, FUT79/, a carcinogenity prediction system /FUT771/ and a system for calculation of physicochemical properties of organic compounds /DAR782/ have been implemented. The expert system in question has an additional feature relative to them: the general nature of the problem and the model formulated permits our program system, with minor changes, to be applied in numerous other fields as well.

Among others, potential application fields are the quality control, geology, town planning, environment protection, all of them dealing frequently with regression models including yes/no variables and their combinations. The fact, that interpretation of the combined variables, a bottleneck of the model-building, could be performed by a relatively short program shows, that logical programming is a powerful tool in constructing small expert systems.

Faltam os títulos

References

BOC65
K. Boček, J. Kopecky, M. Krivucová, D. Vlachová: Experientia 2o 667 /1965/

DAR75
F. Darvas, I. Futó, P. Szeredi, in: Proc. Conf. Comp. Cyb. Methods in Medicine and Biology, p. 413, Ed. D. Muszka, Szeged, Hungary

DAR78
F. Darvas, I. Futó, P. Szeredi: Int. J. of Biomed. Comp. 9 259 /1978/

DAR782
F. Darvas, I. Futó, P. Szeredi, in: Proc. Symp. on Chem.-Struct. - Biol. Act.: Quant. Approaches, Ed. R. Franke, Akademie Verlag, Berlin, 1978.

DAR8o
F. Darvas, J. Röhricht, Z. Budai, B. Bordás, in: Chemical Structure-Biological Activity Relationships, Eds. J. Knoll, F. Darvas, Pergamon Press, London 198o, p. 25.

ELG82
J. Elguero, A. Fruchier, Affinidad 39 548 /1982/

FRE64
S. M. Free, J. W. Wilson: J. Med. Chem. 7 395 /1964/

FUJ71
T. Fujita, G. Ban: J. Med. Chem. 14 148 /1971/

FUT76
I. Futó, P. Szeredi, F. Darvas, in: Proc. Conf. Logique et Base de Données, Toulouse, ONERA, 1977, p. 18.

FUT771
I. Futó, F. Darvas, E. Cholnoky, in: Proc. 2nd Int. Congr. of the J. Neumann Society, Budapest, 1977

FUT79
I. Futó, F. Darvas, P. Szeredi, in: Logic and Data Bases, Ed. J. Minker, Plenum Press, N. Y. 1979

GOL8o
V. E. Golender, A. B. Rozenbliet, in: Drug Design, Vol. IX., p. 299, Ed. E. J. Ariens, Academic Press, N. Y. 198o

HAN63
C. Hansch, R. M. Muir, T. Fujita, P. P. Maloney, F. Geiger, M. Streich: J. Amer. Chem. Soc. 85 2817 /1963/

HAN69
C. Hansch: Accounts Chem. Res. 2 232 /1969/

HAN75
C. Hansch, C. Silipo, E. E. Steller: J. Pharm. Sci. 64 1186 /1975/

KOP65
J. Kopecky, K. Bocek; D. Vlachová: Nature 2o7 981 /1965/

KUB762
H. Kubinyi: J. Med. Chem. 19 587 /1976/

KUB763
H. Kubinyi, O. Kehrhahn: J. Med. Chem. 19 1o4o /1976/

LOP83
A. Lopata, F. Darvas, K. Valkó, Gy. Mikite, E. Jakucs, A. Kis-Tamás; Pestic. Sci., accepted for publication, 1983

MAR78
Y. C. Martin; Quantitative Drug Design. A Critical Introduction. M. Dekker, N. Y. 1978

REG81
J. A. Reggia: Ann. Biomed. Eng. 9 6o5 /1981/

SCH75
L. J. Schaad, R. H. Werner, L. Dillon, L. Field, C. E. Tate: J. Med. Chem. 18 344 /1975/

SHO76
E. H. Shortlife, R. Davis, S. G. Axline, B. G. Buchanan, C. C. Green, N. Cohen: Comp. Biomed. Res. 8 3o3 /1975/

DRA66
N. R. Draper, H. Smith: Applied Regression Analysis. J. Wiley, N. Y. 1966

DEVELOPING EXPERT SYSTEMS BUILDERS IN LOGIC PROGRAMMING

Eugenio Oliveira
Dept. Informatica,
Universidade Nova de Lisboa
Quinta da Torre
2825, Monte da Caparica

ABSTRACT

We intend to develop a set of kits to build Expert Systems using Prolog. Two principal modules, a Knowledge Base acquisition and consultation subsystems are now presented.

Several knowledge representation structures and mixed inference mechanismes are proposed for the sake of system efficiency. Finally some explanation capabilities derived accordingly with used inference methods are also implemented and presented.

.Introduction

Knowledge Based Systems are typical, useful and practical Artificial Intelligence applications.

Knowledge Representation schemas, Problem Solving methods, Natural Language interfaces, Knowledge acquisition capabilities, Plausible reasoning are several important techniques we can find inside AI to build up more intelligent systems to perform expert's knowledge into a great variety of domains.

Knowledge is the very fundamental component of such systems. Nevertheless, if such systems may obey the paradigm of being " Knowledge rich even if they are methods poor ", efficiency and friendliness must not be neglected for the sake of usefulness.

Our experience with Expert Systems (ES) -- Knowledge Based Systems embodying knowledge of one or more experts in a given domain (medicine, geology, ecology, business...) -- gave us some particular insights in such a tradeoff. So, a number of design ideas we now present evolved from past work in ORBI.

ORBI [PER] is an Expert System designed for environmental

resource evaluation, written in PROLOG and running on a PDP 11/23 which gives advice about regions aptitudes and resources. It has a dynamic Knowledge Base entered and modified by experts (not programmers) and supports its decisions with more or less detailed explanations about its reasoning.

One of the fundamental lessons of ORBI development and implementation is PROLOG suitability to encode in a declarative manner structured knowledge about the world (semantic networks, production rules...) as well as the query language, relational database, intermediate interpreters, all of this with the same clear formalism (Horn clause logic).

One of the important drawbacks of most existing systems is that they reflect specific domain particularities loosing all the generality.

Other critical point of such systems is the difficult acquisition of new knowledge and modification of old one directly from experts without the need for computer scientists.

Recent developments show some attempts at generalizing pre-existing ES, trying to present domain independent mechanisms and to be a general framework to deal with at least some classes of worlds.

It is our aim to develop more versatile, powerful and simple Expert Systems Builders using Logic Programming.

.System organization

Our system is able to acquire interactively all the concepts of each new world, to represent them internaly, to relate them, to display them in a comprehensive manner on user's demand. It must have an efficient and versatile procedural behaviour to achieve intended results well enough supported with explanations.

The system can be regarded as two main cooperating modules :

--- Knowledge Base Acquisition Subsystem (KBAS)
--- Consultation Subsystem (CS)

KBAS guides the expert accepting his structured knowledge, individualizes and defines domain concepts, keeps all existent relationships, so entering a complete new world into the system.

.Knowledge Base

Each entity is a triple <concept, attribute, value>. With these entities a conceptual semantic network is built up,whose nodes, corresponding to single concepts (for example "disease"), are expanded on records with several fields (for example meaning, number of attributes...). One of these fields is pointing another tree of concept's attributes each of which with its own characteristics.

We can see this part of Knowledge Base as organized into three layers :

--- Templates, abstracted schemas for concepts's characteristics and rule models.

For example :

```
concept( n. of attributes, attributes names,
         dependencies, contributions, meaning).
```

--- Conceptual network, connecting and naming all particular domain concepts.

For example :

```
therapy( 3, attrbtrp(_,_,_), [disease,pacient],
         [none],[medical advice]).
```

--- Concepts tree, particularizing for each concept all its attributes characteristics. Note that the second argument of the predicate representing a specific concept is a new data structure whose instantiations represent all the attributes characteristics under that concept. After having selected a specific concept predicate, its attributes predicates are directly accessed by means of that second attribute.

For example :

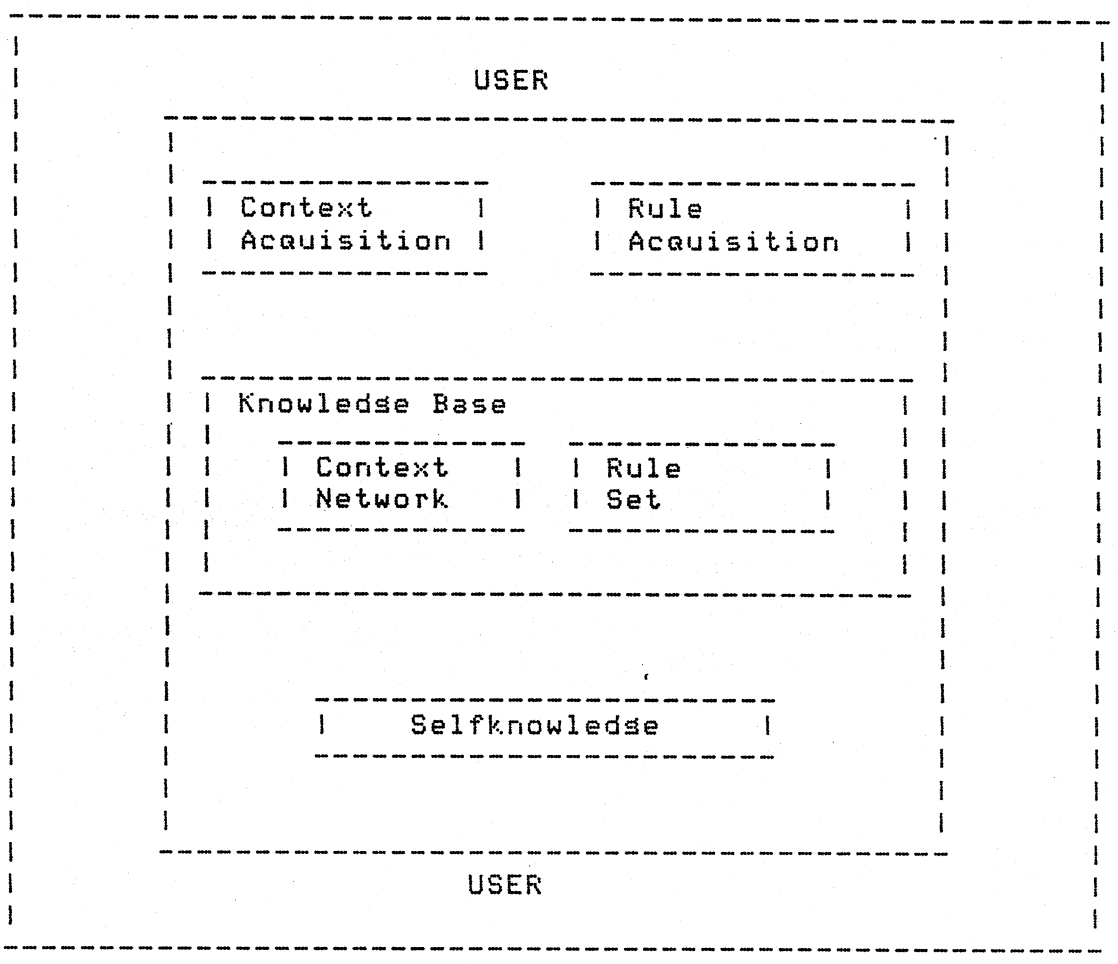
```
attrtrp( 'attribute name', 'attr. unities',
                           'how is obtained').
attrtrp('attribute name',... ).
...
```

All this contextual knowledge is once for all entered by the expert and then it guides the consultation subsystem over the protocol session. It also gives the structure of knowledge which can be consulted by the user.

This feature which is called metaknowledge or selfknowledge represents a kind of introspective capability of knowing about its own knowledge and showing it. Of course that Prolog's

property of programs seen as data facilitates this possibility.

This kind of knowledge archetypes is also important to check rule acquisition. In fact, the other knowledge base major component is a set of Production Rules each of which embodies a chunk of expert domain knowledge, drawing inferences from some concept attribute value to other one. We can display the KBAS module as following :



Rule acquisition is done in a flexible stylized language where several words are recognized as operators (where, if, else, or, and, not...), others as concept attribute's names and its specific values, and finally other ones as functions like (equal, greater,lesser,different...). It is obvious that Prolog non unit clauses (declarativly : 'Head' if 'body') are clearly suitable to represent production rules ('conclusion' if 'premises').

After being checked for syntatic and mostly semantic consistency (regarding contextual network) production rules are precompiled in an internally procedural form and are modularly integrated. A Prolog procedure (a set of Prolog clauses) looks into an intermediate file to where the inputed rule was sent reads it and build the correspondent new clause.

During such an operation optimizations are done to avoid duplication of evaluable predicates into rule's bodies in case of complicated concatenations of boolean operators ("or's" inside "and's", "ored" branches with same concept attribute's name), and so improving rule's fireing efficiency.

Note, however, that they can be displayed again in the same form as they had been entered, by means of a decompilation module which translates them back from internal representation to the more friendly input language.

Rules which shall capture experts knowledge as near as possible its primitive form, must, if necessary, enable several conclusions and complex permises. This is a clear and natural way of trying to prove several conclusions (goals) in a pre-determinate sequence.

When the expert gives such a complex rule he means that those alternative conclusions of the rule are connected and if the first one fails, second shall imediatly be tried and so on. This kind of agregate often corresponds to knowledge organization in expert's head. What I mean is that non--determinism is not always the best way to deal with knowledge representation.

Note that later, if the system keeps track of its successfully fired rules it will know not only which was the right conclusion but also that other ones appearing before in the same rule were tried and failed.

So, rules space is organized as a set of rules subsets (also known as knowledge sources), regarding each concept's attribute, and each rule can either have several alternative conclusions or only one.

This is a very nice improvement. In fact, other well known systems like Emycin [VME], only have very simple rules with one conclusion and a conjunction of single permises.

Our rules can, if necessary, be much more complete as seen in the following example :

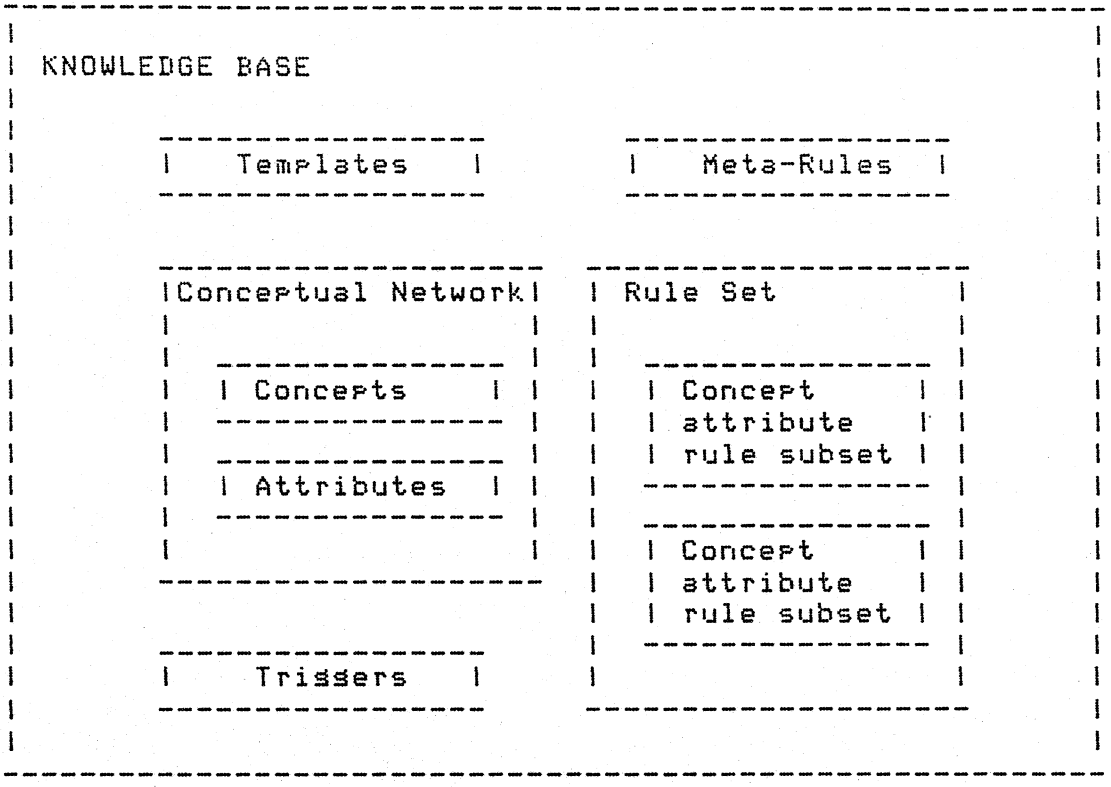
```
disease name = influenza

        if
                syndrome name = headache
                        and
                syndrome duration > two days
                        and ...
                or
                symptom name = feever
                        and
                (symptom intensity > moderate
                                or ...)
                        and ...

else
        disease name = ...

        if ...
```

Rules can also be inspected about theirs components, and concept attributes which contribute for them can also be retrieved, on demand, by searching into theirs bodies.

Several other anotations like rule's name, author and data are also given.

System has a few meta-rules. Meta-rules embodie knowledge about rules themselves. One meta-rule asks the expert if he wants to encode such an ordered alternative conclusions and, if it is the case, instructs him about rule's form.

Knowledge Base is structured as followins :



.Summarizing

When a session besins and if user's (a certain domain expert) password enables him to access knowledge base buildins he can either enter a completly new knowledge base or consult an old one for updatins.

Updates are kept in a separate file to be consulted alternativly or if modifications are definitive the new file will contain the old one already updated.

During knowledge base buildins, the user is on a hierarchical way asked for :

-- concept's names, theirs short mnemonics, theirs mutual relations, number of attributes, meanins.

-- concept attribute's names, possible values, its unities (if mesurable), how shall they be known to the system.

-- Which are the attributes values (if there are any) whose presence is able to directly senerate a set of hypothesis,

expectations to be verified later on. Such an information (called 'triggers') will be responsable for win of efficiency during evaluation process of the consultation session, approaching once more experts way of reasoning.

Simple or complex rules are entered, checked, compiled, retrieved and organized into knowledge sources.

Guided modifications can be done either into the rules or concept network.

If Selfknowledge module is activated all knowledge base information can be accessed (including rules bodies) and clearly presented.

Once again Prolog and its assotiated Horn clause logic it is a very natural formalism to encode knowledge either facts or rules.

.Consultation and Inference mechanisms

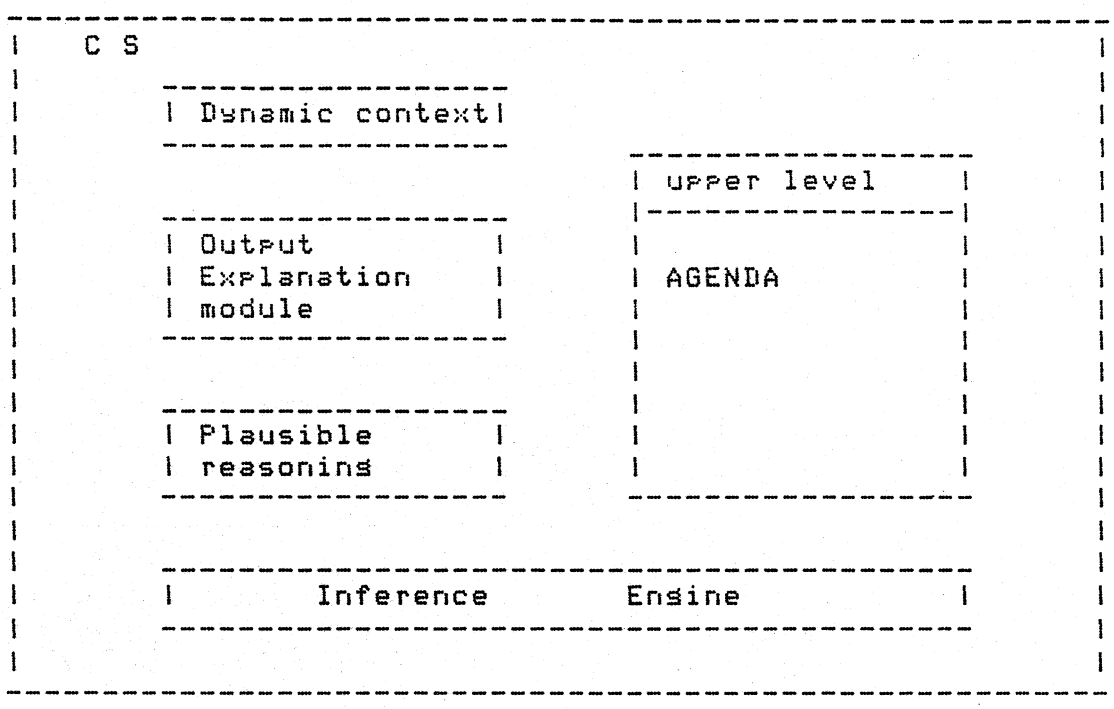
Consultation subsystem, a module under development, uses a selected Knowledge Base (for example a certain medical field), previously build up by means of KBAS module, to interact properly with the user. Note that any user (and not only domain experts) are now able to use Consultation system.

In each session, as result of such an interaction, all needed information is collected. A dynamic context network is built up accordingly to the static one, and the conclusion is reached in an efficient way using production rules selected from respective knowledge sources. Explanations are also obtained.

From KBAS the system already knows possible top goals (ex. therapy, disease). It also knows the set of "triggers" (hypothesis generators), and between them those whose values shall be asked for at the beginins of the session.

So, a protocular session when consultation starts, collects these possible highly discriminatory information.

At that moment inference engine is applied to these two extremes of the space problem (data and top goal) to find the solution.



.Inference methods

Expert systems can be used in a number of applications so diferentiated as diagnosis, plan, design and education among others.

It becomes very difficult to present enough generalized techniques to cover all these possibilities. Nevertheless, several reasoning methods should be available to give versatility to each particular class of systems.

As Prolog is our unique implementation language, the sussested strategy is backward chaining (goal directed) and depth first with backtraking. This is also the strategy used by well known ES like Mycin (Shortliffe) as well as its derived essential system (EMYCIN).

However if search space is very large (namely if the proof tree has a big amount of parallel branches near the root), simple top down becomes inefficient and other search methods must be pursued.

Our inference engine takes advantage of initial data token from the user and of Knowledge Base information about hypothesis generators to prune, earlier, the search space tree. A little cyclic interpreter takes these data, asks KB

for appropriate (matched with that data) 'triggers' and, if they exist, climbs up the tree, guessing some hypothesis and trying to prove them all the possible ways around. If it is the case, these hypothesis (now intermediate conclusions), are asserted in an 'agenda', a globally accessible data structure, and the cycle is repeated with these asserted facts and other possible 'triggers'.

Very many initial possibilities can so be discarded and search space will become more workable with this kind of hierarchical generate and test method.

When this cycle is over the inference engine still keeps the main objective it wants to achieve (top goal) and 'agenda' has all proved intermediate conclusions. At this moment the system can choose between two reasoning methods : forward chaining from asserted data or backward chaining from top goal till it meets assertions in 'agenda ' or in the data base.

'Agenda' has a segmented structure with an individualized upper level. So, as forward chaining proceeds, not every assertion in the 'agenda' is taken into account but only those in upper level, representing nodes nearer top goal, avoiding combinatorial explosion of search paths. In each cycle nodes directly connected with those ones but one step up the tree are tryed to be proved. This implies a reconfiguration of the upper level 'agenda', deleting assertions from which the cycle had started and the assertion of proved new ones. Such a process may continue till top goal is reached if desired.

If backward chaining is choosen, which depends on efficiency considerations, the interpreter looks for the top goal clause, try to prove its body and so on recursively till it meets data or already proved facts on the 'agenda'.

These methods imply, of course, that 'agenda' access mediates each decision step.

.Explanations

Mixing hypothesis generation with forward and backward chaining mechanisms makes explanation task not so easy as if it was only one direction inference (for example top down like in Orbi or Mycin).

During computation all proved steps are carried on a special data structured argument or conveniently asserted in the 'agenda'. They are connected by different constructs depending how they were infered or if they are alternative paths. We keep this executed code, a kind of program's trace, and then we look at it as data to be manipulated. Prolog's

programs declarativity is once more very much appreciated.

An appropriate output procedure deals with these constructs building up an enough understandable output explanation, where we can distinguish between input data, suessed information and step by step infered conclusions.

Already used in Orbi the system will also dispose of another interpreter which discriminates inside rule's body deterministic parts from non-deterministic ones, computing the former and delaying the latter. This interpreter, like other ones into the system, is of course written in Prolog.

An exemple of a compound explanation will be :

Explaned answer for "X" :

"E" is a valid intermediate conclusion because :

"A" was given for you
and "B" is a fact
and to the question "C" you answerd "D"

still another explanation for "E" is :
I already know "F"
and the truth of "F" implies "E"

and finally from "E" I can deduce "X".

.Conclusion

Under development is a kit of programs to build up Expert Systems in several knowledge domains. At present we focused our atention at system architecture and convenient knowledge representation structures, selfknowledge inference mechanisms and explanation capabilities.

Other components like natural language interface and plausible reasoning models will be later on implemented.

We propose to combine several knowledge representation structures as semantic networks and production rules, to use a meta-knowledge module to guide user's consultation, to apply hypothesis generation and bidirectional inference. Composed but understandable explanations are already suggested.

Prolog is our unique implementation language and a very much suitable one.

References

[PER] ORBI- an Expert System for environmental resource evaluation through Natural Language
Pereira,L. ; Oliveira E. ; Sabatier P.
Proceedings First International Logic Programming Conference, Marseille 1982.

[VME] The Emycin manual
Van Melle ; Scott ; Bennet ; Peairs
Department of Computer Science, Stanford University 1981.

Submitted to the Logic Programming Workshop, Portugal, June 83

KBO1: A Knowledge Based Garden Store Assistant

Adrian Walker

IBM Research Laboratory
San Jose, California, USA

Antonio Porto

Universidade Nova de Lisboa
Lisboa, Portugal

ABSTRACT

This paper describes a prototype knowledge based system which answers English questions about some of the products supplied by a garden store. The system answers questions such as

what can I use to kill snails ?

is there anything I can use that will fertilize my lawn ?

what can I use to kill weeds in my lawn in spring ?

does product A kill dandelions in less than 20 days ?

in less than one second each, and it produces a helpful phrase when it cannot answer a question.

The syntax, semantics and knowledge needed by the system are written in the language Prolog. The behavior of the system indicates that Prolog appears to be a good language for the construction of practical knowledge based systems which can answer questions in ordinary English.

1. INTRODUCTION

This paper describes KBO1, a prototype knowledge based system which answers English questions about some of the products supplied by a garden store. The system matches, to a certain extent, the behavior of a helpful, knowledgeable assistant in a store which sells products such as domestic pesticides and weed-killers.

Some questions which produce brief, but useful and accurate answers, are:

what products do you sell ?

what is each product that you sell for ?

what can I use to kill snails ?

is there anything I can use that will fertilize my lawn ?

what can I use to kill weeds in my lawn in spring ?

what can I use to kill weeds around my fence ?

do I need a sprayer to use product A ?

what is the response time of weeds to product A ?

does product A kill dandelions in less than 20 days ?

The present system only answers questions about certain house and garden products. Questions which lie outside this scope, such as

is there a bus stop near here ?

are answered with a sentence such as

I'm sorry, I don't know the word: bus

The system has been written to explore the feasibility of building useful knowledge bases in the programming language Prolog (1, 2). Prolog appears promising as a notation for implementing knowledge based natural language systems, since knowledge rules and grammar-like rules can be written down and executed more or less directly (3, 4, 5).

In the case of our KBO1 system, it was not necessary to write an inference engine or a parser. Both of these items were covered by the use of the built in inference method of Prolog. We have used an efficient implementation of Prolog (7) on an IBM mainframe computer. In terms of coverage of a part of English, our results are promising. In terms of performance, we find that no question takes more than 1 second to answer, even when our computer is heavily loaded with work by other users. Our coverage and performance results are consistent with those reported by (3, 5, 9), namely, that efficient Prolog programs can be written for useful natural language access to knowledge bases.

Section 2 of this paper describes the input-output behavior of the KBO1 system by means of some annotated examples. Section 3 outlines the internal design of the syntactic and semantic components of the English interface of KBO1, while Section 4 describes the knowledge base. Section 5 consists of conclusions and directions for future work.

2. INPUT-OUTPUT BEHAVIOR OF KBO1

A user of the KBO1 system types English questions on a keyboard, and gets answers on a screen. In the following example session, a question starts after a prompt symbol ">>" and ends with a question mark "?". The answer produced by the system follows the question. Comments which we have added appear in parentheses after answers.

```
>>
what products do you sell ?

product A
product B
product C
product D
product E
product F
```

(the system has simply listed the products.
for examples with more products, it would print
just the kinds of products)

```
>>
what do you know about ?

product A kill weed
product B kill pest
product C kill pest
product D kill insect
product E kill insect
product F kill weed
product F fertilize lawn
```

(the system has listed the products in its knowledge base,
with one or more intended uses for each product. English
output generation is minimal)

```
>>
what is each product that you sell for ?

product A kill annualweed
product A kill perennialweed

product B kill gardenpest

product C kill gardenpest

product D kill crawlinginsect
product D kill flyinginsect
product D kill hoppinginsect
```

product E kill crawlinginsect
product E kill flyinginsect
product E kill hoppinginsect

product F kill annualweed
product F kill perennialweed
product F fertilize lawn

(this question has put more emphasis on intended use than the last, so the system has answered in more detail)

>>
when should I use product F ?

spring
fall

>>
what can I use to kill snails ?

product B
product C

>>
what can I use to kill snails on my vegetables ?

product C

(the product product B is not for use on food, and so is not suggested in this answer)

>>
what can I use to kill weeds ?

product A
product F

(these are the suitable products known to the system)

>>
what are the weeds ?

annualweed
perennialweed

(the system has listed, without much detail, the kinds of weeds)

>>
what are weeds ?

sorry, I don't know

(we have reached a limit of the knowledge in the system)

>>
what are the perennial weeds ?

bahiagrass
bull thistle
clover
dandelion
kentucky bluegrass
poison ivy
poison oak
yellow oxalis
oxalis

(the system has understood this as a request for a detailed answer)

>>
is there anything I can use that will fertilize my lawn ?

product F

(a suitable product is suggested)

>>
what can I use to kill weeds in my lawn in spring ?

product F

(the same product can be used)

>>
what can I use to kill weeds around my fence ?

product A

(product F is for lawns, but elsewhere we can use something stronger)

>>
do I need a sprayer to use product A ?

yes

>>
can I use product A without a sprayer ?

sorry, no

>>
what are the precautions I should follow in using product A ?

avoid contact with eyes
keep children and pets away till spray has dried

avoid contamination of food
if accidental spray then wash off with water

>>
what is the response time of weeds to product A ?

annualweed 5 days
perennialweed 15 days

>>
what is the response time of each product that kills annual weeds ?

product A 5 days
product F unknown

>>
what can I use on dandelions ?

product A
product F

>>
what can I use on dandelions that will kill them in less than 20 days ?

product A

>>
does product A kill dandelions in less than 20 days ?

yes

>>
does product A kill dandelions in less than 2 days ?

sorry, no

This session shows that the KBO1 system has considerable knowledge of the properties of a few products and their intended uses. As with all knowledge bases known to us, there are limits to the domain which is covered. However, when a question cannot be answered, the user sees reasonable replies such as "I'm sorry, I don't know", or "I don't understand the word: bus". Although the domain of competence is much smaller than that of most adult people, these phrases carry about the same information as an immediate reply from a person who does not know the answer to a question.

It would appear worthwhile to extend KBO1 so that it would explain its answers when asked to do so. For example, it would be helpful to know why product B cannot be used to kill snails on vegetables. The techniques described in (8) could be used to do this.

In this section, we have described the input-output behavior of KBO1. The next section outlines the design of the English interface, while section 4 describes the knowledge base.

3. THE ENGLISH INTERFACE

Our two main goals in designing the English interface were simplicity and modularity. We wanted to keep the interface simple so as to be able to get a system working within a short time. On the other hand, we made the interface modular so that it can form a basis more elaborate English input processing. The interface uses full dictionary lookup for every word in the sentence. Thus it is distinct from a 'keyword' interface, in which some words may simply be ignored.

Our main simplification is to make lexical analysis deterministic. This means that only one morphological token is associated with each word, an assumption which does not hold in general, but which which turns out not to be a serious limitation in our present application. This simplifying assumption allows us to complete the lexical analysis before starting a syntactic parse. In a later version of the interface we would expect to drop the deterministic assumption, but to retain modularity and efficiency by using coroutining techniques in Prolog (6).

Some other design desicisions which we made to keep the English inteface simple were as follows.

We only deal with fairly simple ellipsis, namely an elided subject in a conjunction of verb phrases. We feel that any serious treatment of ellipsis would need to manipulate information from several sentences in a dialog. The present system works one sentence at a time.

We do not generate a syntactic tree. Rather, we construct a semantic representation directly during parsing. This is efficient, since semantic checks are made early in the anaylsis of a sentence, helping to prune unfruitful parses.

We do not treat extraposition, although some common cases of left extraposition can be handled by an easy extension of the present system.

We also do not make a syntactic check of gender and number agreement, since we have found that it can only be used constructively in some rare cases in which it may disambiguate an attachment problem. (Even then, semantic checks may be enough.) In the same vein, no analysis is made of verb tense.

We made the English interface modular by clearly separating a number of functional components, and by classifying each component as either specific to our present application, or general. The main components are: a lexical analyser, a dictionary, a syntactic parser, semantic rules, and an output package. The lexical analyser and the syntactic parser are general purpose. The dictionary and the semantic rules each have a general subcomponent and an application-specific subcomponent, while the output package is application specific.

The lexical analyzer reads a question from the terminal, groups the characters into words, and looks up the words in the dictionary to find the corresponding morphological tokens (e.g. noun, verb, preposition).

As mentioned above, the dictionary consists of a general and a specific part. The general part contains entries that are likely to be needed in any application in English, such as articles and the forms of the verb 'to be'. The specific part contains entries for items such as product names for the KBO1 application.

The syntactic analyzer consists of a set of rules, written in the manner of a definite clause grammar (4). The rules are executed top-down by Prolog's built-in inference mechanism, hence there is no distinct syntactic parser component in KBO1. The bodies of the grammar rules contain the usual calls to nonterminal symbols, and also contain calls to the semantic rules that construct a representation of the meaning of the sentence which is being parsed. If a fruitless parse is being attempted, the calls to the semantic rules will fail, causing a new parse to be sought before too much effort has been wasted.

The function of the semantic rules is to translate groups of syntactic items into a semantic representation of a question. The semantic representation, which we describe in detail below, is a Prolog statement roughly equivalent to 'the set of all X such that p(X) is true in the knowledge base', where X and p may be structured terms. In particular, p may contain further set-formation terms. The general part of the semantic rules component treats items such as quantification and the verb 'to be' which would be needed in most applications, while the specific part deals with items such as the kinds of objects to which it is reasonable to apply domestic chemical products.

The output component uses the information which is retrieved from the knowledge base, together with syntactic information about the question which caused the retrieval, to generate the answer that is displayed on the screen.

Since KBO1 is designed to answer questions, the syntactic component of the English interface only accepts questions. These may be wh-questions, (such as 'what...', 'when...', 'how...' etc.) or nexus questions ('does...', 'is there...', etc). Verbs are accepted both in the active and the passive forms, and they can be followed by any number of complements. Nouns can be preceded by any number of adjectives, and double nouns are accepted (e.g. 'poison ivy'). Nouns can have simple complements ('persistence of product A') or double complements ('response of bluegrass to product A'). The syntax covers the usual articles and prepositions, a few pronouns ('you', 'them', 'anything', ...) and some auxiliary verb forms ('can', 'should', ...). Relative clauses are accepted, and there can be conjunctions of relative clauses, verb phrases, or verb complements.

The concepts represented inside the parser are of two types, which we call Entities and Properties. A complete meaning representation, in terms of Entities and Properties, is constructed from the morphological token stream by the syntactic and semantic rules. Entities are either Objects, or sets of Objects, the latter being represented by 'set(Q, O)', where Q is a quantifier ('each', or 'all') and O is an Object.

Objects can be named, or can be defined. A named Object is represented as 'T:X', where T is the name of the type of the Object. If X is a free variable, then we say that the Object is abstract, that is, it is an unspecified Object of type T. If X has a value, then we say that the Object is concrete, and that X is its name. Thus 'perennialweed:X' is an abstract Object, while 'perennialweed:clover' is a concrete Object.

Defined Objects have the form

Abstract_Object ! Property

(read 'Abstract_Object such that Property') where the Abstract_Object is T:X and X appears in the Property.

A simple Property is just a Prolog predicate. Properties may also be quantified, in the form

for(Qantifier, Object, Property)

Conjunctions of Properties are also Properties.

Here are some concepts and their internal representations:

product_A	item:product_A
a weed	weed:X
the weeds	set(each, weed:X)

all weed killers

```
set(all, item:I ! use(I, kill-weed:W, S))
```

persistence of each product

```
time:T ! for(each, item:I, use(I, *, persistence(T)))
```

There are semantic predicates that link concepts to form new concepts. They link subject and verb, verb and object, verb and complement, and adjective and noun. The domain-independent part of the definition of these predicates deals with the handling of quantification and the verb 'to be', while the domain-dependent part contains only quantifier-free definitions.

After a syntactic and semantic parse succeeds in producing a data strucure corresponding to an input question, the data structure is transformed into a Prolog query which can be applied to the knowledge base. The essence of the transformation is to insert a call to the Prolog meta-predicate 'all', which computes the set of all items which satisfy a given property.

For example, for the question

'what is the response time of weeds to product A ?'

the query

```
set(response):R !

   all(T,use(product_A,kill-weed:any,response(T)), R)
```

is generated. This query, when executed against the knowledge base, retrieves the answer to the original question and binds the answer to the variable R.

Special care was taken to give informative, rather than yes-no, answers to nexus questions. To see that this is essential, rather than simply desirable, consider the question

'do you sell anything that kills bluegrass ?'

Very few people would be satisfied with only the answer 'yes', so the system generates the query

```
set(item):I ! all(P, use(P, kill-weed:bluegrass, M), I).
```

which retrieves a list of suitable products.

On the other hand, a yes-no question such as

'does product A kill dandelions in less than 20 days ?'

is translated into the query .

```
yesno ! use(product_A, kill-weed:dandelion, response(T)) &

typedIt(T ,20.days)
```

This section has described the design of the parts of the KBO1 system that translate a question in English into a query for the knowledge base. The next section describes the knowledge base.

4. THE KNOWLEDGE BASE

The last section described the mapping of an English question into either a form

yesno ! p(X)

or a form

set(items):X ! p(X)

where p(X) is, in general, an abitrary Prolog goal expression. This section describes the underlying knowledge base which provides answers to the mapped queries.

The knowledge base consists of two components, both of which are domain-specific. The first component contains an is-a hierarchy, while the second contains knowledge about the products and how they may be used.

The hierarchy contains assertions such as

```
setname(weed)
setname(annualweed)
weed(annualweed)
annualweed(bluegrass)
```

together with an immediate membership predicate 'mem', a transitive membership predicate 'member', and an 'isa' predicate. Thus mem(bluegrass, annualweed) holds, while mem(bluegrass, weed) fails, but member(bluegrass, weed) holds. Similarly isa(bluegrass, bluegrass), isa(bluegrass, annualweed), and isa(bluegrass, weed) all hold. The 'mem' predicate is also written in infix form as ':', e.g.

mem(bluegrass, annualweed)

is written annualweed:bluegrass.

The type-hierarchy is actually a directed acyclic graph rather than a tree, as there are statements such as

```
homepest(fly)
flyinginsect(fly)
```

The hierarchy allows questions to be answered at an appropriate level of detail. For example

'what can I kill with product A ?'

yields the answer 'weeds', while

'what weeds can I kill with product A ?'

yields

annual weeds
perennial weeds

The main body of knowledge about the products is stored in the second component of the knowledge base. It consists of

(i) an input component, which maps sub-queries of the form

use(Subject, Verb-Object, Modifiers)

into calls to some basic clauses about the products,

(ii) the basic clauses themselves, e.g.

can_use(product_A, kill, weed:Y:Z) <- weed:Y:Z

and,

(iii) an output component which contains information about which kinds of questions (yesno, set) require which kinds of answer format.

The knowledge base is used as follows. A query such as

'what should I use in spring to kill weeds in my lawn ?'

is presented to the knowledge base as

```
set(item):X !
      all(S, use(S, kill-weed:any,
          environment(plant:lawn).season(spring)), X).
```

Here all(S, use(S,....), X) returns in X the set of all subjects S such that one can use S for the indicated purpose. The call

use(S, kill-weed:any, environment(plant:lawn).season(spring))

is mapped by the knowledge base into

```
can_use(S, kill, weed:any) &
environment(S, kill, weed:any, plant:lawn) &
season(S, kill, weed:any, spring)
```

Thus a subject S is retrieved if it can be used to kill some Object O, the Object O matches weed:any in the type hierarchy, and the modifiers environment and season are satisfied.

The knowledge base contains basic clauses such as

```
can_use(product_F, kill, weed:Y:Z) <- weed:Y:Z
environment(product_F, kill, weed:*, plant:lawn).
season(product_F, *, *, spring).
```

which cause the original call to use(S, ...) to succeed with S = product_F.

As mentioned above, the general form of an internal query to the knowledge base is

```
use(Subject, Verb-Object, Modifiers)
```

where Modifiers is a list made up from some of the predicates: environment, season, persistence, response (e.g. response time of a weed to a weed-killer), ingredient, assume, precaution, directions (i.e. directions for use, and how (precautions and directions). The modifiers in the list may be negated, and the list denotes a conjunct.

The predicates for how, precaution and direction are special in that they store English text which can be retrieved, but which cannot be checked in detail. Thus one can ask 'what are the directions for using product A ?' which yields the query

```
set(direction):S !
      all(D, use(product_A, V-O, direction(D), S)
```

and the answer

```
apply with hand trigger sprayer
one application kills most weeds
less effective if rain within 24 hours
```

However the question 'does product A kill most weeds in one application ?' yields 'Sorry, I don't understand'.

The remaining predicates can be used either for retrieval or for checking, and there is some overlap between these and the retrieval-only predicates. Thus the question 'what vegetables can I spray with product D ?' yields the query

```
set(vegetable):S !
      all(V, use(product_D, V-O,
            environment(vegetable:V).
            assume(equipment(sprayer))), S)
```

which retrieves into S the answer 'any'.

One can also ask the checking question 'can tomatoes be sprayed with product D ?' which yields a yesno query similar to the one above, but for

```
environment(vegetable:tomato)
```

and leads to the answer 'yes'.

The use of retrieval-only predicates to store English sentences is mainly a matter of convenience. If detailed questions about, e.g., directions for

the use of a product, were expected, then the knowledge could be moved to predicates which could be used both for retrieval and checking.

To summarize, the knowledge base consists of a hierarchy together with specific knowledge about products and their uses. An incoming internal query from the English interface is transformed into a Prolog goal, the goal is executed against the knowledge, and the result is sent to the output component of the system.

5. CONCLUSIONS AND DIRECTIONS FOR FURTHER WORK

The KBO1 system is at present a prototype. Our experience in bringing it to its present level of behavior indicates that Prolog is well-matched to the task of building a knowledge based natural language system. The system answers non-trivial questions in under one second of real time on an IBM mainframe computer.

While certain simplifications were made in order to build a demonstrateable system in a short time, the English language interface performs full dictionary lookup and parsing. The system was built in a modular manner, and we have separated the reusable parts from the domain dependent parts.

Adding new words and their meanings the system is rather straightforward. Many extensions to the syntactic parser could be made without having to change other parts of the system. For example. we could improve the present treatment of left extraposition just by modifying the parser. In fact, 'what' is already treated like an extraposed noun, and 'when' like an extraposed complement.

A major improvement would be to handle anaphora, mainly ellipsis and pronoun reference beyond the scope of one sentence. However, this is still a research area needing much work. An interesting point is that people sometimes make outside references not only to a previous question, but also to previous answers. To resolve such references, we must have access to a representation of the previous answers.

Another interesting enhancement would be the treatment of cardinal and fuzzy quantifiers. This would require that we modify the semantic rules that handle quantification, and that we define the equivalent of the 'all' meta-predicate for the new quantifiers. A nice possibility is a generalized 'all' meta-predicate with an extra argument which would impose conditins on the number of solutions.

In summary, the direct representation of syntax, semantics and knowledge in the language Prolog appears to be a good approach to the construction of useful knowledge based systems which can answer questions in ordinary English.

6. REFERENCES

(1) Clocksin, W. F. and C. S. Mellish, Programming in Prolog. Springer-Verlag, 1982.

(2) Kowalski, R. Logic for Problem Solving. North-Holland Publishing Co., 1979.

(3) McCord, M. Using slots and modifiers in logic grammars for natural language. Artificial Intelligence 18, (1982), 327-367.

(4) Pereira, F. C. N. and D. H. D. Warren. Definite clause grammars for language analysis - a survey of the formalism and a comparison with augmented transition networks. Artificial Intelligence 13, 1980, 231-278.

(5) Pereira, L. M., P. Sabatier and E. Oliveira. Orbi: an expert system for environmental resources evaluation through natural language. Proc. 1st Int. Logic Programming Conference, University of Marseilles, France, 1982, 200-209.

(6) Porto, A. Epilog: A language for extended programming in logic. Proc. 1st Int. Logic Programming Conference, University of Marseilles, France, 1982, 31-37.

(7) Roberts, G. M. An implementation of Prolog. M.S. thesis, Department of Computer Science, University of Waterloo, 1977.

(8) Walker, A. Prolog/EX1: An inference engine which explains both yes and no answers. Proc. 8th Int. Joint Conf. on Artificial Intelligence, Karlsruhe, to appear.

(9) Warren D. H. D. and F. C. N. Pereira. An efficient easily adaptable system for interpreting natural language queries. DAI Research Paper No. 155, Department of Artificial Intelligence, University of Edinburgh, 1981.

Database Management, Knowledge Base Management and Expert System Development in PROLOG

Kamran Parsaye

*Computer Science Research International**

ABSTRACT: The programming language PROLOG suggests a natural way of combining programming and deductive database queries by treating both programs and data as assertions in a database. We explore some issues in the implementation of databases and expert systems in PROLOG. We show that some simple extensions to PROLOG will allow for the convergence of many concepts from relational databases and expert systems into a uniform formalism for the management of both data and knowledge.

1. Introduction

The programming language PROLOG**, has been an interesting step in modern language design. By its nature of design, PROLOG *includes* a database and is hence a suitable language for database applications, particularly relational databases. Due to its symbolic nature and deductive capabilities, PROLOG is also a suitable language for expert systems implementations. Thus PROLOG seems a good candidate language for implementing both databases and expert systems.

In this paper we explore some issues which arise in the implementation of databases and expert systems in PROLOG. We show that some simple extensions to PROLOG will allow for the convergence of many concepts from relational databases and expert systems into a single formalism. This formalism can be used to approach both *database management* and *knowledge-base management* in a uniform manner.

Our extensions to PROLOG are, however, intended to preserve the flavor of PROLOG as a language. For instance, we show that the concept of functional dependency in relational databases is essentially equivalent to some PROLOG "cuts", that integrity constraints may simply be treated as PROLOG assertions, and that explanations and transparent reasoning in expert systems can be viewed as PROLOG execution traces. Many of the issues presented here grew out of the work on EDD (Expert Database Designer), a PROLOG based expert system for database design [Parsaye 82].

This paper is organized as follows: In section 1.1 we give a brief description of the language PROLOG. In section 2 we relate standard relational database concepts and terminology with PROLOG. We suggest an extension to PROLOG mode declarations, show the relationship between cuts and functional dependencies, and show how integrity constraints can be treated. Section 3 is devoted to PROLOG optimization issues for large database applications. We present a classification scheme for PROLOG clauses and propose the "independence assumption" for optimization. We also suggest how the notions of transactions

*) Author's Address: Computer Science Research International, 6420 Wilshire Blvd., Suite 2000, Los Angeles, CA 90048.

**) In this paper PROLOG essentially refers to the language originally defined by [Colmerauer 75] and implemented by [Warren 77].

and serializability can be easily introduced into PROLOG. In section 4 we focus on expert system applications. We propose a uniform view of database and knowledge-base management and illustrate two closely related approaches to knowledge representation in PROLOG. We also show how features such as explanations and transparent reasoning can be naturally programmed in PROLOG.

1.1 The Language PROLOG

In the context of this paper, it is particularly interesting to compare the development of PROLOG as a language to similar developments in data models and database languages.

Early database systems, e.g. IMS or CODASYL, use data structures such as trees or networks to store data. Users of IMS store and retrieve data by explicit insertion and retrieval operations which act upon tree structures, and in this sense deal with a *structure oriented language*. On the other hand more recent database systems, e.g. relational databases, hide the underlying data structures and implementation details from the user, and present associations and relationships in a non-navigational form.

Similarly, in programming languages such as FORTRAN, LISP or ADA one has to create data structures such as arrays, lists or stacks, store his data within these structures and later retrieve the data by navigational searches. On the other hand, in a *database oriented language*, such as PROLOG, the user can be unaware of the underlying implementation methods used for storing much of his data, and simply ask for data items to be stored and retrieved, just as he would ask a relational database system for storage and retrieval of data.

Software development in PROLOG can thus be mostly based on *"programming by assertion and query"* [Robinson 80], rather than by insertion and searchs of data structures. Moreover, the style of PROLOG programming is *declarative*, in the sense that a predicate (procedure) definition explicitly includes both the input and output parameters. Thus in PROLOG the distinction between input parameters and output parameters is much less prominent than in other languages, as seen by the examples below.

We now present a very brief and informal description of PROLOG, proceeding mostly by example. A detailed and comprehensive description of the language can be found in [Clocksin & Mellish 81], or [Pereira, Pereira & Warren 79].

The basic building blocks of PROLOG programs are *clauses*. A clause in PROLOG is a predicate name, called a *functor*, with some *arguments*. For instance

```
father(john, mary).
square(3, 9).
```

are clauses, where 'father' and 'square' are functors and 'john', 'mary', 3 and 9 are arguments.

Arguments may be constants or variables, and conventionally, non-numeric constants are denoted by lower case letters, while variables must start with uppercase letters, e.g. as in father(X, mary).

In PROLOG clauses can be *asserted* to be true, in which case they are included in the PROLOG *"database"*. The PROLOG database contains all facts which are asserted to be true. For instance,

assert(father(john, mary)).

will include father(john, mary) in the database and

retract(father(john, mary)).

will remove it.

In PROLOG, clauses are used to make *sentences*. A sentence in PROLOG may be a simple *unit* clause, such as father(john, mary), or it may involve the *conditional construct* denoted by " :- ", and better understood as "if".

For example, the conditional sentence

parent(X, Y) :- mother(X,Y) .

means that "for all X and Y", parent(X, Y) is true if mother(X, Y) is true. Thus essentially " A :- B " means that A is logically implied by B.*

Clauses on the right hand side of a " :- " can be joined together by "and" and "or" constructs denoted by "," and ";" respectively, as in

parent(X, Y) :- mother(X,Y) ; father(X,Y) .

which means that "for all X and Y", parent(X, Y) is true if either mother(X, Y) is true "or" father(X, Y) is true.

Let us make two simple technical notes here. First that sentences in PROLOG must end with a period. Second that due to the universal quantification above, the range of each variable in PROLOG is essentially a sentence, i.e. two occurances of the same variable name within two sentences are totally unrelated. The PROLOG compiler will internally rename variables to avoid conflicts.

Unlike equational programming languages, such as OBJ [Goguen & Tardo 79] or HOPE [Burstall et al. 80], PROLOG allows variables on the right hand side of a conditional which do not appear on the left hand side. Such variables are intended to be *existentially quantified*. For instance the sentence

grandfather(X, Y) :- father(X, Z), parent(Z, Y).

means that "for all X and Y", X is the grandfather of Y if "*there exists*" some Z in the database, such that X is the father of Z and Z is a parent of Y.

In PROLOG, conditional clauses may be stored in the database just as data are, i.e. programs are really treated as data in a database. This uniform view of both programs and data as items in a high level database is perhaps the major reason for the elegance of the PROLOG programming style.

Once one has adapted this database view of programming, one may naturally wonder about queries to the database. Simple queries may relate to simple facts such as: "Is father(john, mary) true in the database?", which may simply require a look up in the database, However, one may also ask more complex queries.

We generally refer to an attempt to answer a query in the PROLOG database as an attempt to *satisfy a goal* (or to *prove a goal*). For instance, in the example above "father(john, mary)" is the goal, and it can be satisfactorily proved if father(john, mary) has been asserted in the database.

One may also try to prove goals with variables, in which case PROLOG will try its best to find a match for the variables to satisfy the goal. For instance, an attempt to prove "father(X, mary)" will succeed provided that the condition (X = john) is true. Note that this is not an assignment (PROLOG is assignment free),

*) Logically speaking, PROLOG sentences are Horn Clauses [Horn 51].

but a binding of a variable to a value as in pure LISP. Such bindings are discarded upon the completion of the query.

Now, how about conditional clauses? Since the interpretation of the conditional construct " :- " is that the right hand side logically implies the left hand side, the validity of the left hand side can be established by proving the right hand side. This new goal may itself in turn be part of a conditional clause,..., and so on. Thus execution of programs in PROLOG essentially consists of attempts to establish the validity of goals, by chains of pattern matching on asserted clauses.

To prove a goal PROLOG searches its database for a clause that would *match* the goal, by using the process of *unification* [Robinson 65]. If a conditional clause whose left hand side matches the goal is found, PROLOG tries to satisfy the set of goals on the right hand side of ":-" in a left to right order. If no matching clause can be found, *failure* will be reported.

It must be noted that PROLOG includes no explicit negation symbol, and negation is essentially treated as *unprovability*, i.e. the failure to establish a goal from a set of axioms [Clark 79]. This closely resembles the closed world assumption [Reiter 78].

If PROLOG does not succeed in establishing a goal in a chain of deductive goals at a first try, it will *backtrack*, i.e. go to the last goal it had proved and try to satisfy it in a different way. For instance, suppose that we have the sentences (or program)

parent(X, Y) :- mother(X,Y) ; father(X,Y). (*)

grandfather(X, Y) :- parent(Z, Y), father(X, Z). (**)

and that the following facts have also been asserted:

father(john, mary).

father(paul, john).

mother(jennifer, mary).

Then to prove "grandfather(X, mary)", by using (*) and (**) above, first the goal "parent(Z, mary)" will be tried. This in turn will result in an attempt to prove "mother(Z, mary)" and will succeed with (Z = jennifer). Then, going back to the "grandfather" clause again, the next goal in the conjunction should be proved. So "father(jennifer, Y)" will be tried and will fail. At this point PROLOG will go back (i.e. backtrack), discard the assumption (Z = jennifer) and try to prove "parent(Z, Y)" again. This time (Z = john) will result, after trying "father(Z, mary)". Then the eventual binding (Y = paul) will be returned, after trying "father(X, john)".

Let us note that in the grandfather "program" here there are no explicit *input* or *output* parameters, i.e. one may either invoke grandfather(X, mary), or grandfather(paul, Y). This style of *declarative programming* in PROLOG can often be used to great advantage to develop software very rapidly. However, if a parameter in a program is always intended to be an input or output, the compiler can be signaled to generate optimized code by including *mode declarations* of the form

:-mode square-root(+, -).

which means that the square root function is never intended to be used to multiply a number by itself. Thus the user has the choice of running a program in both directions or not, as he sees fit.

On one hand, the series of steps taken by the PROLOG compiler in proving a goal essentially amount to *deduction*. On the other hand an attempt to prove a goal "father(X, Y)" can also be looked at as a procedure call to the predicate father. Thus the use of the term "logic programming" is quite apt here.

Calls in PROLOG can also be recursive, as in

connected(X, Y) :- edge(X, Z), connected(Z, Y).

which deals with connectivity in graphs described in terms of edges. The PROLOG compiler [Warren 77] uses tail recursion optimization to great advantage in such cases.

Now, for expression evaluation. In the author's opinion, one of the most inconvenient features of symbolic languages such as LISP has been the relation between quotation and evaluation. The PROLOG approach to evaluation is exactly the opposite of LISP, i.e. evaluation does not take place until it is forced to. This is specially relevant to arithmetic expressions and removes the need for quotes. Thus (2 + 3) can be evaluated to 5 when the need arises, by using the PROLOG infix operator "is", i.e. "X is (2 + 3)" binds X to 5. However, again note that this is not assignment.

Finally, one other feature of PROLOG which we need to mention is the "cut", denoted by "!". The cut is used to control backtracking in PROLOG. It is just treated as a goal itself, and can be used in any conjunction or disjunction of goals. Any attempt to satisfy "!" will succeed immediately for the first time, but will signal the compiler never to try it again. In fact an attempt to "retry" a cut will fail the parent goal invoking it, e.g. in

a(X) :- b(X,Y), !, c(Y,Z), d(Z).

backtracking can take place between c and d, but PROLOG will never backtrack to b. The cut can thus be used to gain efficiency and control in programs.

Many more examples of PROLOG programs, and a more detailed description of the language and its use may be found in in [Clocksin & Mellish 81], or [Pereira, Pereira & Warren 79].

2. PROLOG and Relational Databases

It is well known that relational databases can be viewed as logical predicates [Nicolas 77]. Essentially, each table in a relational database can be considered as the 'extensional' specification of a predicate. Each PROLOG predicate on the other hand, can be viewed as the 'intensional' specification of a relation or table. Moreover, it is also well known that most 'assertions', dependencies and integrity constraints in relational databases can be expressed as Horn Clauses [Fagin 80], which are essentially PROLOG sentences. Thus there is a natural correspondence between PROLOG and relational databases.

However, there are differences between existing relational concepts and PROLOG. In the next 3 sections we outline some of these differences and show, how with some simple extensions, they can be reconciled.

2.1 Schemas and Types

Relational databases usually rely on a typed system of logic and include schema information which determines the type and domain of attributes. PROLOG currently lacks these notions and relies on an untyped system of logic.

However, as [Nicolas 78] shows, an untyped system of logic can be easily

used to represent typed logic. For instance, the typed assertion

V XεINT p(X)

can be represented as the untyped sentence

V X (integer(X) & p(X)) ,

where & denotes conjunction.

The addition of schema and type information to PROLOG without affecting the flavor of the language is quite easy. PROLOG already includes mode declarations of the form:

:-mode employee(+, +, -).

which, for selected predicates, can be used to signal the compiler as to which parameters are intended as input and output.

To declare schemas, we suggest adding schema declarations of the form

:-schema employee(name, age, salary).

Similarly, we can add type information of the form

:-type employee(string[12], integer[3], integer[7]).

However, we believe that the inclusion of type information need not be mandatory and the user should be allowed to exclude type declarations for small relations, or when he sees fit.

The gain from having the declarations is two fold: on one hand they can be used for type checking and error detection, on the other they can be used by the compiler to achieve considerable enhancement in performance.

We feel that a major shortcoming of most current PROLOG implementations is that the compiler can not be informed that the argument to a square root function is intended to be an integer (rather than an arbitrary list), or that a social security number is a string of 9 digits. In most large database applications one needs to specify some type information and fixed length record sizes. We believe that before PROLOG can be used in a "real" large database application it should be extended to allow for the inclusion of type information within programs.

2.2 Functional Dependencies

PROLOG currently includes no notions of dependencies and normalization so far. These concepts were introduced into relational database theory since they are needed for design and for the avoidance of update anomalies. We believe that these concepts should be introduced into PROLOG in order to make it suitable for database applications. Moreover, in section 3.3 we show how functional dependencies can sometimes be used for optimization purposes.

Functional dependencies are simple enough to preserve the elegance of the PROLOG programming style. However, we feel that the addition of more complex dependencies, such as MVD's [Zaniolo 78] [Fagin 78] or EMVD's [Parker & Parsaye 80], may add an unnecessary amount of complexity to PROLOG programs.

Functional dependency information can be added to PROLOG in a manner similar to the type and schema information. However, interestingly enough, not only can this concept be incorporated into PROLOG quite naturally, but it gives rise to a different style of PROLOG programming.

In relational database terminology [Armstrong 77], the existence of a functional dependency A->B in a schema p(A,B) means that for each A there is only one B such that p(A,B) is true, e.g. X->Y in father(X, Y) means that each child

has at most one father.

We suggest the introduction of functional dependencies into PROLOG programs by declarations of the form

:- dependency(A->B) in p(A, B).

:- dependency(AB->C) in q(A,B,C).

At first a functional dependency may seem similar to a PROLOG construct of the form

..., p(A,B), !, ...

which fails the parent goal invoking p(A,B) if any goal following the cut fails. If a binding for A is supplied by the parent goal the cut is essentially equivalent to having the dependency (A->B) in p(A,B). In this case after failing p(A, B) once, one could not hope to find a new value for B by retrying p(A,B).*

However, if B is supplied by the parent goal and A is to be found by invoking p(A, B) then the the cut and the dependency are not equivalent, since the cut still forces the search to end. We feel that sometimes this use of cuts is against the general PROLOG philosophy that programs can be run in both directions when desired.

In general there has been a good deal of dissatisfaction with cuts in PROLOG anyway. We suggest that in many cases functional dependencies would be a much better alternative to cuts. Functional dependencies can often be used to write "cut-free", but efficient PROLOG programs, by directing the execution of programs in a manner which is dependent on the mode of procedure calls. Thus with the above functional dependency, in evaluating

q(A,B) :- ..., p(A, B), ...

there is an implicit cut after p(A, B) in the evaluation of q(a,B), but not in the evaluation of q(A,b). Moreover, note that the two sided declaration

:- dependency(A<->B) in p(A,B).

can be used to achieve a symmetric effect.

Of course, there are cases where one wishes to terminate the search after one unsuccessful attempt even though there is no dependency, in which case a cut will have to be used. However, this generally reduces the elegance and transparency of the "cut-free" PROLOG programming style.

2.3 Integrity Constraints

Enforcing database style integrity constraints expressed by Horn Clauses is very natural in PROLOG and is essentially a form of integrity enforcement by query modification [Stonebraker 75].

Clauses are usually added to the PROLOG database by the predicate 'assert', which adds almost anything to the database, without any integrity checks. To enforce integrity, we suggest the use of a predicate 'add' to assert facts which are subject to an integrity check. 'Add' is itself defined in PROLOG by

add(C) :- not(invalid(C)), assert(C).**

Conditions which should not be allowed in the database are indicated by the predicate 'invalid'. Thus, to enforce an integrity constraint on a predicate we add an assertion about invalidity. For instance, assume that we wish to enforce

*) Provided the integrity of the database has been preserved, as discussed in section 2.3.
**) Where 'not' denotes negation as unprovability.

the fact that an employee whose age is less than 19 can not earn over 100,000, i.e. that in

employee(Name, Age, Salary)

Salary should be less than 100,000 if Age is less than 19. We can simply add the assertion

invalid(employee(Name, Age, Salary)) :- (Age < 19) , (Salary > 100,000).

Thus the assertion

add(employee(johnson, 18, 120,000))

will fail, since

invalid(employee(johnson, 18, 120,000))

will succeed.

Functional dependencies are a special form of integrity constraint and will hence have to be enforced during addition of new data. A functional dependency (A->B) in p(A,B) can be enforced by simply adding the constraint

invalid(p(A, B)) :- p(X,B), not(eq(X,A)),

where 'eq' is defined by eq(X,X).

One may also wish to deal with the validity of responses, i.e. to ensure that returned values are consistent. Then one can define

return(A) :- A, not(invalid(A)).

to return results. Updates to the database can then be treated by combining additions and deletions.

The discussion above is aimed at integrity constraints that are usually placed on relational databases, i.e. constraints which essentially deal with unit clauses. We feel that enforcing constraints on non-unit clauses will often involve such a great deal of computation as to make it practically non-feasible.

3. Large PROLOG Databases

Having considered some high level database and language issues, we now focus on large database implementation and optimization issues relating to PROLOG.

Currently, all implementations of PROLOG either reside totally in core or rely on virtual memory. This proves to be sufficient for general programming and very small databases, but is certainly inadequate for serious database applications. However, we believe that with a suitable implementation strategy PROLOG can also be successfully used in conjunction with very large databases.

Moreover, since large databases are almost always shared by many users, we also need to consider PROLOG in a multiuser database context. We shall deal with these issues in the next three sections.

3.1 The Independence Assumption.

Much of the appeal of PROLOG has been the unification of the concepts of programming and querying into a single discipline by treating programs and data in a unified manner at the user level. However, while the user may be unaware of this distinction, we feel that for optimization purposes, a PROLOG implementor should separate these facts and deal with them accordingly.

Clauses in PROLOG can be classified into three categories:

a) *Non-unit Clauses*, i.e. clauses with both a left and a right hand side, e.g. clauses of the form p(A,B) :- q(A, C), r(C, B, X).

b) *Unit-Clauses with variables*, i.e. clauses with no right hand side, but with a variable argument, e.g. clauses of the form p(a,X).

c) *Ground-Unit Clauses*, i.e. clauses with no right hand side, and with no variable arguments, e.g. clauses of the form p(a, b, c).

Almost all of the information stored in current relational databases is of type c), while PROLOG 'programs' mostly contain clauses of types a) and b).

Currently most PROLOG implementations store and retrieve data by directly accessing a predicate's clause and (sometimes) hashing on one or more of the arguments. Moreover, almost all implementations use the same hashing method for clauses of class a), b) and c). In most large database applications this is simply an unacceptable implementation strategy since the size of and frequency of access and updates to data can be very different from the corresponding size and frequency for programs. Hence different hashing and indexing methods for these different categories of clauses are called for.

At first it may seem that the presence of a large number of 'database facts' of type c) and 'programs' of type a) for a given functor name can cause a problem since it may not be clear what form of hashing or indexing should be used for that functor name. However, we suggest that this need not be the case, and that the above classification can be used to implement large deductive databases more efficiently by making the following independence assumption:

For each given functor name, it is unlikely that there are a large number of Non-unit clauses and a large number of Ground-unit clauses at the same time. It is also unlikely that there are a large number of Unit-clauses with variables for any given functor name.

Assuming that Non-unit clauses are essentially 'programs' and Ground-Unit clauses are mostly 'data', the independence assumption means that programs and data are usually referred to with different functor names. The user may, if he wishes, indicate whether a functor name will be used for large database applications by a declaration of the form

:- largedata(employee(name, social-security-no, salary)).

Different hashing and indexing schemes may thus be used for these different classes. It would also be desirable to provide indices not only on the first argument but on other arguments of a predicate as specified by the user with a declaration of the form

:- index(B), index(C) in p(A, B, C).

which provide extra indices for B and C.

In this context, an interesting form of indexing for use in conjunction with deductive database systems has recently been proposed by [Lloyd 82].

3.2 Transactions, Concurrency.

Currently PROLOG is really only for single user personal databases, and includes no notions of transactions and concurrency control. Large databases are almost always accessed by more than one user, and there is a need for controlling the interleaving of the different user's programs in order to preserve the consistency of the database.

If PROLOG is to be used in large database applications, there will be need for sharing parts of databases between different PROLOG programs. This is not

directly related to expert system issues, but a PROLOG based expert system may need to access a shared database, say of patient medical records.

There will also be a need for including some form of transaction specification facility in PROLOG. There is also a need for the modification of most PROLOG implementations so that they would provide better interaction facilities with operating systems.

The introduction of transactions and concurrency control would require that some specified parts of a program be indicated as "atomic" actions, which are not interleaved with other programs. This is really a very simple point, and we are including it mostly for the sake of completeness.

To illustrate the concept of atomicity, consider a PROLOG transaction which performs transfers between accounts, i.e. the predicate

```
transfer(Account1, Account2, Amount) :-
        balance(Account1, X), balance(Account2, Y),
        Z is (X + Amount), W is (Y - Amount),
        retract(balance(Account1, X)), retract(balance(Account2, Y)).
        add(balance(Account1, Z)), add(balance(account2, W)).
```

The interleaving of the execution of this predicate with another user program such as

```
printsum(Account1, Account2) :-
        balance(Account1, X), balance(Account2, Y),
        W is (X + Y), print(W).
```

may result in inconsistent results. Thus the user needs to specify that he wishes 'transfer' to be an atomic action on the shared database.

We suggest adding simple declarations of the form

```
:- atomic(transfer(account, account, amount)).
```

to specify that a predicate should be implemented as an atomic transaction. The method of concurrency control can of course be left to the database operating system.

3.3 Implementing the 'Setof' Predicate

Some PROLOG implementations provide a predicate 'setof' which retrieves all instances of variables satisfying a predicate (or conjunction of predicates), e.g.

```
setof(X, (p(X, a, Y), q(Y,b), r(Y,c)), L)
```

retrieves into L all X for which p, q and r are true. Of course in many situations the order in which the predicates are evaluated can make a big difference. This form of conjunctive query optimization occurs quite frequently in database applications. Both System R [Astrahan et al. 76] and CHAT-80 [Warren & Pereira 81] deal with this issue by looking up relation sizes and reordering conjunctions.

We feel that the need for introducing this optimization into CHAT-80 is simply an indication of the fact that such a feature is missing from the basic PROLOG implementations. If PROLOG is to be used as a 'database' language, such feature would be necessary. It would not be hard to add such a feature essentially as CHAT-80 has implemented it.

Moreover, sometimes it might be possible to do even more optimization by using functional dependencies. A user can specify the order of the evaluation of conjuncts in his programs if he wishes, but any given order is not optimized for different modes of procedure calls. Again due to the PROLOG philosophy that

programs should be runnable in both directions it would be a good idea to allow the optimization to vary with the mode of the procedure call. Often it is possible to optimize in these situations if a functional dependency is known, e.g. if we know that

dependency(A->B) in q(A,B)

then in the evaluation of

setof(X, (p(X, Y), q(a, Y)), L)

it would usually be advantageous to evaluate q before p. This is also helpful in CHAT-80 like applications.

4. Some Expert System Issues

So far, we have discussed the appeal of PROLOG in database applications. Due to its symbolic nature and deductive capabilities, PROLOG is also a suitable vehicle for implementing expert systems. In the past few years, PROLOG has been the major language for expert system implementations in Europe. Some such systems, e.g. [Pereira & Porto 82], [Pereira et al. 82], [Darvas et al. 79], [Markusz 80] among others, offer encouraging results.

In the next sections we discuss the appeal of PROLOG's uniform approach to data and programs in expert system applications and and show how issues such as knowledge representation, explanations, transparent reasoning and inheritance can be dealt with.

4.1 Databases and Knowledge-bases

Currently, most expert systems dealing with databases have two distinct notions of *database management* and *knowledge-base management* [Davis & Lenat 82]. Often, the interaction between the knowledge-base and the database is not as smooth and well coordinated as one would wish.

As we have discussed before, PROLOG treats both programs and data in a uniform way. In expert systems applications, this can be looked upon as a single view of both 'data' and 'knowledge'. We suggest that this single view of both data and knowledge can be used to approach both database management and knowledge-base management in a uniform and elegant manner.

Looking back at the history of computing systems, one can view this as part of a general trend towards the development of very high level interfaces for interactive systems. The user interfaces of the computing systems of the 1960's were essentially based on the notion of *file management*, while since the early 1970's there has been a distinct trend towards high level *database management*. As [Ohsuga 82] points out, the user interfaces of the computing systems of the late 1980's and beyond are very likely to be mostly based on *knowledge-bases*. This signifies a general trend towards a uniform and high level style of interactive computing based on intelligent knowledge-based interfaces. We believe that PROLOG's uniform view of data and knowledge is a good basis for this gradual movement towards this form of knowledge-based interactive computing.

PROLOG is particularly useful in expert system applications which need to use large databases in one of the following ways:

a) They need to interact with large amounts of 'data' stored in databases, e.g. as in the RX system [Blum 82] which bases its inferences on a large database of medical case histories.

b) They need to use a database to store a large amount of 'knowledge' in

terms of a large number of rules which pertain to an area of expertise, e.g. expert systems which deal with a manufacturing environment [FGCS 81].

Of course, there are also many cases where both of the above conditions are satisfied. The advantage of using PROLOG in such applications is that the unified manner in which PROLOG approaches both data and programs (and in this case 'knowledge') results in a uniformity of design which facilitates the interaction between the human expert, the knowledge engineer and the expert system. As [Buchanan 79] points out, uniformity in design and representation is of great value in the development of expert systems.

We feel that in due course of time, most computing environments will be eventually liberated from the concept of a file system and will exclusively deal with unified databases and knowledge-bases. We also believe that due to its uniformity of approach, PROLOG is an excellent vehicle for this transition.

4.2 Knowledge Representation

Since PROLOG programs are essentially a subset of the sentences of first order logic, a natural knowledge representation method in PROLOG is a "logic flavored" knowledge representation method similar to MRS [Genesereth 81b]. Such representation has many advantages, but as we shall discuss later, it need not necessarily be the sole conceptual representation method for expert systems developed in PROLOG.

In the logical approach, the world is viewed in terms of 'predicates', and knowledge is essentially captured in terms of logical implications, i.e. production system like rules, or 'if then else' conditions. Such representation is in a way similar to the methods used by R1 [McDermott 80]. PROLOG sentences offer a convenient way of representing such rules, both in terms of 'deep' and 'surface' rules [Hart 82].

For instance, this form of representation is quite useful in the development of expert systems for diagnostic applications [King 82]. SUBTLE [Genesereth et al. 81a] uses an essentially similar approach. For example, a basic and general rule about the malfunctioning of structured components, say in an instrument diagnosis expert system, would be

```
malfunction(X) :- subcomponent(X, Y), malfunction(Y).
```

where the subcomponent information can itself be included in the database, as shown for example by

```
subcomponent(instrument, sensor).
subcomponent(instrument, connector).
subcomponent(instrument, display).
```

Specific structure relating to connectivity can be represented by assertions of the form

```
connector-input(X) :- sensor-output(X).
display-input(X) :- connector-output(X).
```

On the other hand, assertions of the form

```
malfunction(connector) :-
    connector-input(X), connector-output(Y), not(eq(X, Y)).
```

can be used to reflect the input/output relationships for the components.

In this example, note how easy it is to deal with the knowledge-base about the structure of the components just as one deals with a relational database containing parts and components information. Moreover, sometimes in the

course of diagnosis and repair of an instrument, the expert system may wish to gather information about the availability of "field replaceable units" from a common shared database. This can again be handled quite naturally by using the framework suggested in the previous sections.

However, the logical knowledge representation method need not be the only knowledge representation method used in conjunction with PROLOG. We feel that the "None for all, but any for some" truism of programming languages also applies to knowledge representation methods, i.e. that there is *no* knowledge representation method that is good for *all* applications, but that *any* knowledge representation method is perhaps good for *some* application. This suggests that one may use PROLOG in conjunction with different knowledge representation methods in different applications. We must, however, point out that the differences in these approaches are essentially conceptual and in many cases one approach may easily be translated into the other without much difficulty.

Another approach to knowledge representation would be a semantic network like approach, e.g. as suggested in [Brachman 80]. However, as [Deliyanni & Kowalski 79] point out, PROLOG's logical form can be closely linked to semantic network based knowledge representation techniques [Findler 80]. Moreover, in database applications, semantic network like representations may also be viewed as using some form of Entities and Relationships. EDD [Parsaye 82] uniformly uses the Entity-Relationship model [Chen 76] both for database schema design and for capturing the knowledge used in the design process by viewing Entity-Relationship diagrams as semantic networks.*

An example of a situation in which an Entity-Relationship like representation is intuitively appealing is in expert systems for office automation or in database design. As [Deliyanni & Kowalski 79] showed, in such cases one can simply capture the *schema* structure of the Entity-Relationship diagram by assertions of the form

```
relationship(employment, department, employee).
attribute(employee, name).
attribute(employee, social-security-number).
attribute(employee, department-number).
```

which reflect the fact that "employment", is a relationship between the entities "department" and "employee", and that "name", "social-security-number" and "department-number" are attributes of the entity employee. The translation of this representation to a logical form is very similar to the translation of Entity-Relationship diagrams into relational schemas, i.e. it involves the transformation of entities into relation names and attributes into arguments. For instance, the entity "employee" will be transformed into a relation schema

```
:-schema employee(name, social-security-number, department-number).
```

which can later be used to store information such as

```
employee(jones, 558 53 8973, departmet-4).
```

Thus, as is the case with Entity-Relationship diagrams and relational schemas, the logical and network-like representation methods can easily be translated into each other.

---

*) The fact that with very simple modifications, semantic networks diagrams can be easily transformed into Entity-Relationship diagrams has been part of computer science folklore for some time now.

Another issue that is sometimes quite important in knowledge representation is that of *subtypes* and *inheritance*. For instance, it is sometimes very useful to a user to deal with both "employees" and "managers", and record the fact that each manager is also an employee. There is a lot that can be said about such *polymorphic type structures* in theoretical terms [Parsaye 81], [Mac Queen 82], but in most practical cases these issues are quite simple to deal with.

As mentioned in section 2.1, types can be captured in untyped logic by the use of conjunctions, and thus such properties can easily be included in PROLOG programs by assertions of the form

employee(X) :- manager(X).

which specifies that each manager is an employee.

The inclusion of such conjunctions in PROLOG programs is no more easy or difficult than explicit type declarations for variables in a typed language, but this approach provides the flexibility of having or not having the types as desired.

4.3 Explaining Facts and Deductions

It is well known that relational databases can be viewed as logical theories [Nicolas 77], [Jacobs 81]. With this view, almost all data stored in, and queries posed to, current relational database systems deal with facts which are *ground literal* logical assertions or Ground-Unit PROLOG clauses. Such sentences correspond to what might be termed *who and what* facts and queries, e.g. "Who is the manager of department X" or "What is the salary of the oldest employee".

In expert system applications, 'knowledge' is captured in terms of facts which pertain to some form of expertise and need to be represented as non-ground literal clauses, i.e. Non-Unit clause sentences in PROLOG. Queries corresponding to such facts might be termed *how and why* questions, e.g. "Why did you recommend antibiotics for this patient", or "How did you know that this patient has diabetese".

Such queries are important since in the development of expert systems, it is often necessary to query the system about the knowledge used, and the series of deductive steps taken, in a deduction. This form of *transparent reasoning*, i.e. the ability of the expert system to explain and justify its actions and derivations is of utmost importance in the development of expert systems; without it the gradual enrichment of a simple set of rules into a non-trivial knowledge base would be almost impossible.

In such cases, it is not only necessary to explain the method of deduction and the knowledge used in the derivation of the answer, but to record *why* some piece of knowledge is in the database. For instance, it is usually necessary to record the actual patient case history which results in the addition of a rule to a MYCIN like system in order to facilitate future debugging [Shortliffe 76].

Once again, by invoking the uniformity of PROLOG's approach to knowledge and data, we suggest that explanations pertaining to both data and knowledge may be treated in a uniform manner. The basic idea is rather simple: each derivation in PROLOG essentially has the form of a proof tree whose leaves correspond to 'basic facts' or data, while the rest of the nodes reflect the structure of the proof.

The basic facts (i.e. the leaves) are obtained by some empirical means, e.g. laboratory tests, physicians observations, etc. The justification for these facts can be stored in terms of assertions in the PROLOG database itself, by using

assertions like justification(fact, reason), which record a basic reason for a basic fact, e.g.

:- justification(blood-count(johnson, 130), "test on 11/7/82").

Then the predicate "justify" can be used to justify basic facts by

justify(X) :- justification(X, Y), print(Y).

Such method may also be used for justifying the addition of non-unit clauses to the PROLOG database, e.g.

:- justification(rule 133, "patient case history 173").

Moreover, the steps involved in the deduction are essentially those steps involved in pattern matching and unification. Most PROLOG implementations offer debugging facilities which allow the user to trace the steps in the execution corresponding to a certain predicate. We suggest that similar technique can also be used in explanations, e.g. suppose we have

```
grandfather(X, Y) :- father(X, Z), parent(Z, Y).
parent(X, Y) :- mother(X,Y) ; father(X,Y).
father(john, mary).
mother(mary, paul).
```

A first level explanation of "grandfather(john, paul)" can be obtained by following the steps of the unification, i.e.

grandfather(john, paul) *since* father(john, mary), parent(mary, paul).

A further level of explanation may then be obtained by

parent(mary, paul) *since* mother(mary, paul).

Now let "trace(X, Y)" give Y as the top level goals which were used in the derivation of X. The predicate "justify" can then be extended to the trace and be used to give explanations by using "explain", where

```
explain(X) :- justify(X).
explain(X) :- trace(X, Y), explain(Y).
explain(X ',' Y) :- explain(X), explain(Y).
```

Another interesting issue is to ask "*why not*" questions, e.g. "Why is not john the father of mary?". A simple answer to this can be that this fact is non-existent in the database, but sometimes there may be need for the display of partial deductions that fail. This is quite interesting to program in PROLOG, and is left to the reader as an exercise.

There are of course many other issues that need to be dealt with in the context of multi-level explanations. A number of these issues are discussed in [Swartout 81], and a good deal more work remains to be done on the subject.

5. Conclusions

We have shown how PROLOG can be used to arrive at a uniform and high level approach to both database management and knowledge-base management. We have also pointed out the appeal of this single approach to the management of both data and knowledge in expert system applications. As a language, PROLOG holds a lot of promise. We believe that with the advent of architectures more suited to its implementation [FGCS 81], PROLOG will become a dominant force in computing in the 1980's.

6. Acknowledgements

I wish to thank David Warren and Fernando Pereira of SRI, Stott Parker of UCLA, Edward Katz, Heinz Breu, Robert Fraley and Martin Liu of Hewlett Packard, Robert Blum and Gio Wiederhold of Stanford University, Jonathan King of Symantec and Antonio Porto of the University of Lisbon for helpful comments and discussions. Some of this work was performed while the author was at the Computer Research Center, Hewlett Packard Laboratories.

References

[Armstrong 74]

Dependency Structures of Database Relationships, Proceedings of the 1974 IFIP Congress, Amsterdam.

[Astrahan et al. 76]

System R : A Relational Approach to Data Management, ACM Transactions on Database Systems, Vol 1, No 2.

[Blum 82]

The RX Project, Computer Science Department, Stanford University.

[Brachman 80]

Knowledge Representation in KLONE, in [Findler 80]

[Buchanan 81]

Research on Expert Systems, Heuristic Programming Project Report, Stanford University.

[Burstall et al. 80]

HOPE, An Experimental Functional Programming Language, Proceedings of the 1980 LISP Conference, Stanford University.

[Chen 76]

The Entity Relationship Model, ACM Transactions on Database Systems, Vol 1, No 1.

[Clark 78]

Negation as Failure, in [Gallier & Minker 78].

[Clark 80]

PROLOG, A language for Implementing Expert Systems, in 'Machine Intelligence 10', edited by P. Hayes and D. Michie.

[Clocksin & Mellish 81]

Programming in PROLOG, Springer Verlarg.

[Colmerauer 75]

Les Grammaires de Metamorphose, Groupe d'Intelligence Artificialle, Marsille-Luminy, 1975, reprinted in Lecture Notes in Computer Science Vol. 63.

[Dahl 82]

On Database Systems Development through Logic, ACM Transactions on Database Systems, Vol 7. No 1.

[Deliyanni & Kowalski 79]

Logic and Semantic Networks, Communications of the ACM, Vol 22. No 3.

[Darvas, et al. 79]

A Logic Based Program for Predicting Drug Interactions, International Journal of Biomedical Computing, Vol 9.

[Davis & Lenat 82].

Knowledge-Based Systems in Artificial Intelligence, McGraw Hill, New York.

[Fagin 78]

A New Normal Form for Relational Schemas, Research Report, IBM San Jose Research Laboratory.

[Fagin 80]

Implicational Dependencies, Proceedings of the 1980 ACM Conference on Foundations of Computer Science.

[Forgy 81]

OPS5 User's Manual, Department of Computer Science, Carnegie-Mellon University.

[FGCS 81]

Proceedings of the Fifth Generation Computer Systems Conference, Tokyo, Japan.

[Findler 80]

Associative Networks, North Holland.

[Furukawa 81]

Problem Solving and Inference Mechanisms, in [FGCS 81].

[Gallier & Minker 78]

Logic and Databases, Plenum Press, New York, 1978.

[Genesereth et al. 81a]

SUBTLE Reference Manual, Computer Science Department, Stanford University.

[Genesereth et al. 81b]

MRS Reference Manual, Computer Science Department, Stanford University.

[Goguen & Tardo 79]

An Introduction to OBJ, Proceedings of IEEE Conference on Reliable Software, Boston.

[Hart 82]

The Future of Artificial Intelligence, Artificial Intelligence Technical Report, Fairchild Laboratories.

[Horn 51]

On Sentences which are True of Direct Unions of Algebras, Journal of Symbolic Logic, Vol. 16.

[King 82]

Knowledge-Based Diagnosis and Repair, Internal Memorandum, Hewlett Packard Laboratories.

[Kitagawa 82]

Japan's Annual Reviews in Computer Science and Technologies, North Holland.

[Kowalski 79]

Logic for Problem Solving, North Holland.

[Kowalski 81]

Logic as a Database Language, Department of Computing, Imperial College London.

[Jacobs 81]

Applications of Logic to Databases, Department of Computer Science, University of Maryland.

[Lloyd 81]

Implementing Clause Indexing in Deductive Database Systems, Technical Report, Department of Computer Science, University of Melbourne.

[Mac Queen 82]

A Polymorphic Type System, Technical Report, Bell Laboratories.

[Markusz 1980]

Applications of PROLOG in Many-storied Panel House Design, Proceedings of the 1980 Logic Programming Conference, Hungary.

[Mc Dermott 80]

R1 : A Rule-Based Configurer of Computer Systems, Technical Report, Computer Science Department, Carnegie-Mellon University.

[Michie 80]

Expert Systems in the Micro-electronic Age, Edinburgh University Press.

[Nicolas 78]

Logic and Databases, in [Gallier & Minker 78].

[Ohsuga 82]

Knowledge-Based Systems as a New Interactive Computer System of the Next Generation, in [Kitagawa 82].

[Parker & Parsaye 80]

Inferences Involving Embedded Multi-Valued Dependencies and Transitive Dependencies, Proceedings of the 1980 ACM SIGMOD Conference.

[Parsaye 81]

Higher Order Abstract Data Types, Ph.D. Dissertation, Computer Science Department, UCLA.

[Parsaye 82]

EDD, an Expert System for Database Design, Internal Report, Computer Research Center, Hewlett Packard Laboratories.

[Pereira 82]

Can Drawing be Liberated from the Von Neuman Style?, Technical Report, SRI International.

[Pereira, Pereira & Warren 77]

DEC-10 PROLOG users Guide, Department of Artificial Intelligence, University of Edinburgh. [Pereira et al. 82] Orbi

[Pereira & Porto 82]

A PROLOG Implementation of a Large System on a Small Machine, Proceedings of the 1982 Logic Programming Conference, Marseille.

[Reiter 78]

On Closed-world Databases, In [Gallier & Minker 78].

[Robinson 65]

A Machine-oriented Logic Based on the Resolution Principle, Journal of the ACM, Vol. 12, No. 1.

[Robinson 80]

Programming by Assertion and Query, in [Michie 80].

[Shortliffe 76]

Computer Based Medical Consultations: MYCIN, Elsevier, New York.

[Stonebraker 75]

Implementation of Integrity Constraints on Views by Query Modification, Proceedings of the 1975 ACM SIGMOD Conference.

[Suwa 81]

Knowledge Base Mechanisms, in [FGCS 81].

[Swartout 81]

Explaining and Justifying Expert Consultation Programs, Proceedings of the 7th IJCAI.

[Warren 77]

Implementing PROLOG - Compiling Predicate Logic Programs, Research Reports 39 and 40, Department of Artificial Intelligence, University of Edinburgh.

[Warren 81]

Efficient Processing of Interactive Relational Database Queries Expressed in Logic, Proceedings of the 1981 VLDB.

[Warren, Pereira & Pereira 77]

PROLOG, The Language and Its Implementation Compared with LISP, Proceedings of the ACM Symposium on AI and Programming Languages.

[Warren & Pereira 81]

The CHAT-80 System, Technical Report, Department of Computer Science, University of Edinburgh.

[Zaniolo 78]

Studies on Relational Databases, Ph.D. Dissertation, Department of Computer Science, UCLA.

A database support system for PROLOG

Jan Chomicki
Wlodzimierz Grudzinski

Institute of Informatics
Warsaw University
PKiN
POB 1210
00-901 Warsaw, Poland

1.Introduction

In the current literature PROLOG is often connected with databases, eg (Warren 81), (Lloyd 82). That intuition is, in our opinion, correct. Nevertheless, two main points have been missed: how to organize large PROLOG databases and why they should be superior to the conventional, eg relational, ones. In this paper, we concentrate on the former question. We start by discussing the advantages of PROLOG over the relational model of data . Then we outline simple PROLOG solutions to several database problems. However, some of the most difficult issues of database management do not depend on the language used for defining and manipulating databases.

We describe a database support system for PROLOG implemented at the Institute of Informatics, Warsaw University. According to the taxonomies of (Lloyd 82), our efforts may be described as an interpreted approach, providing (currently)limited database management system capabilities.

The system is primarily meant for the storage, retrieval and modification of a form of PROLOG clauses. Unit clauses designated by PROLOG are managed by the support system which stores them on disk. Other clauses are treated in the standard way by the PROLOG interpreter. When a clause from disk is required, the interpreter issues a query to the support system. Then the system performs preliminary unification and returns all the clauses possibly matching the query(one at a time). The final unification and the binding of variables are performed by the interpreter.

To cope with growing files the database organization and access method is dynamic and based on extendible hashing (Fagin 79), augmented to allow partial-match retrieval (Lloyd 80). A general method has been developed for handling incomplete information (Chomicki 83). This method is used for storing, retrieving and modifying unit PROLOG clauses with variables.

2.PROLOG compared to relational languages

2.1.Relational model of data

Base relations are represented in PROLOG as procedures consisting only of unit clauses.

Domains are not directly mapped to the constructs of the language. They may be defined as unary (not necessarily base) relations, but then unit clauses of a n-ary relation should be augmented by the calls to the procedures defining its domains, so the clauses are no more unit. Or, to achieve more restrictive typing, domains may be treated as functors. But then, for the matching to succeed, queries should submit appropriate domain (functor) names.

Attributes are implicit in the order of relation columns. Their naming for further reference may be locally

(within a clause) obtained by introducing appropriately named PROLOG variables.

Keys(unique tuple identifiers) are in no way supported by PROLOG and, if required, should be defined as integrity constraints(cf section 2.5).

Throughout this paper, we use the original PROLOG notation (Roussel 75). Predicate names are in uppercase, variable names - in lowercase. The goal(procedure head) of the clause is preceded by the "+" character, the premisses - by the "-" character.

2.2.Relational algebra

As the following examples show, PROLOG easily supports relational algebra (Ullman 80) operators: selection, projection and join.

Example 1.

Let R be a binary relation with two numeric attributes: A and B.

Selection:

$\sigma_{A=5}(R)$	+R1(5,x) -R(5,x).
$\sigma_{A=B}(R)$	+R2(x,x) -R(x,x).
$\sigma_{5<A}(R)$	+R3(x,y) -R(x,y) -LESS(5,x).

Projection:

$\pi_A(R)$	+R4(x) -R(x,y).

Equijoin:

$R \bowtie S$	+R5(x,y,z) -R(x,y) -S(y,z).

Conjunction of selection conditions may be expressed by their enumeration in clause premisses and disjunction - by clause variants. Hence in PROLOG the selection condition is in disjunctive normal form.

Now consider set-theoretic operators: union and difference. (or equivalently (Ullman 80): union and division). Union is straightforwardly modelled by clause variants. To define set difference we should resort to some form of negation.

Example 2.

Suppose we write down the difference as

+Q(x,y) -R(x,y) -NOT(S(x,y)).

If no clause defining R contains variables, the argument of NOT will be ground and NOT will be evaluated correctly in the standard way.

2.3.Nulls

The difficulty in defining relational algebra operators in the presence of null(undefined) values has been recognized for some time already: (Codd 79), (Vassiliou 80). There are two ways of representing nulls in PROLOG: as variables or as ground terms.

If null is represented as an unbound variable, the semantics of stored null values ('missing', 'any', 'not applicable')may be enforced by restricting the result of the query (by the predicates differentiating several sorts of nulls). Furthermore, the standard predicates, eg EQUAL or LESS, should be extended to capture properties of different nulls.

If nulls are represented as ground terms, a null matches only either a variable or itself in the query. This disallows the interpretation of a null as an 'any' value matching all the values.

In our opinion, different nulls should be differently represented. More general nulls, eg 'missing' or 'any', expressing possible relevance of the information in a clause to many queries, should be handled as variables. More specific ones, eg 'not applicable', matching no value in a query except itself and a variable, may be treated like a

ground term. Various null values and their semantics are described in: (ANSI 75), (Vassiliou 80), (Zaniolo 82). Note that representing null value as a variable may lead to well known problems with negation in PROLOG: (Clark 79), (Naish 83).

Example 3.

Returning to Example 2, let us define a clause +R(5,g) with the variable g standing for the 'any' null value.
Now writing

+Q(x,y) -R(x,y) -NOT(S(x,y)).

fails to produce the desired result: "all the elements of R which are not in S".
To see that, it is sufficient to make both procedures R(and S) consist only of one unit clause: +R(5,g). (and +S(5,2).).
The query: "is Q non-empty? ", expressed as

-Q(u,w)

fails instead of producing a positive answer.

The above effect may be partly remedied by extending NOT, like EQUAL or LESS, to capture the semantics of nulls.

2.4. Views

Precisely in the same way as queries (relational algebra expressions), database views may be defined in PROLOG. However it is unclear how to do view updates. Currently in PROLOG, modifying the view has no effect on the underlying base relations. Each updating user must access base relations. So PROLOG views do not fulfill their fundamental role of protection mechanisms, hiding information from users. An explicit translation of view updates to the updates of the underlying base relations is required. That translation is determined by the semantics of relations and attributes involved. Sevaral strategies have been proposed in: (Dayal 82), (Bancilhon 81), (Paolini 82), (Siklossy 82). A general mechanism should allow the definition of procedures updating base relations and triggered by view updates (Shipman 81). Such a mechanism is outlined in section 2.5.

It is rather obvious that PROLOG clauses, containing eg recursive calls, are more general than relational views. Nevertheless it is an open problem whether the strategies for updating relational views may be generalized to arbitrary PROLOG views.

2.5. Integrity constraints

Furthermore, it is not well known how to impose integrity constraints in PROLOG. The constraints assert about the consistency of the database, so they should be defined at some meta-PROLOG level. The only solution of that problem we know of was proposed in (Bowen 81). It requires major changes in the PROLOG interpreter. As it supports arbitrary assertions expressed in first order logic, the computational complexity of its implementation would be enormous. We think of developing a less powerful but simpler and more efficient method, outlined below.

Pattern-directed procedure invocation in PROLOG may be used to solve the problems of integrity constraints enforcement, view updates and general triggers. We define two database modification operators: INSERT(clause) and DELETE(clause). They are hidden from the user who sees only "safe" (consistency preserving) operators: INCLUDE(clause) and EXCLUDE(clause) defined as

+INCLUDE(c) -INSERTSAFE(c) -INSERT(c).
+EXCLUDE(c) -DELETESAFE(c) -DELETE(c).

Both INSERTSAFE and DELETESAFE are defined as conjunctions of individual integrity constraints:

+INSERTSAFE(c) -IS1(c) ... -ISk(c).
+DELETESAFE(c) -DS1(c) ... DSp(c).

The problem is how to get the individual integrity constraints. They may be generated manually from informal specifications.

Example 4.
In the relation R first attribute functionally determines the second attribute.
+IS1(R(x,y)) -R(x,y1) -NOT(EQUAL(y,y1)) -/ -FAIL.
+IS1(R(x,y)).

Example 5.
The domain of the first attribute of S contains the domain of the second attribute of R.
+IS2(R(x,y)) -S(y).
+IS2(S(y)).
+DS2(S(y)) -R(x,y) -/ -FAIL.
+DS2(S(y)).
+DS2(R(x,y)).

As may be seen from the above examples, one of the pair of constraints on insertion (deletion) is often tautologically true and may be omitted in the definition of INSERTSAFE (DELETESAFE). However we do not address here further issues of optimizing constraint checking, referring the reader to (Nicolas 82) and (Blaustein 81).

Triggers (Eswaran 76) provide an interesting alternative (or complement) to assertions. In tead of checking the constraints, we may prefer to correct their violations. Now

+INCLUDE1(c) -INSERTTRIGGER(c) -INCLUDE(c).
+EXCLUDE1(c) -DELETETRIGGER(c) -EXCLUDE(c).

and

+INSERTTRIGGER(c) -IT1(c) ... -ITm(c).
+DELETETRIGGER(c) -DT1(c) ... -DTn(c).

The execution of a trigger may obviate integrity checking as seen below.

Example 6.
All the assertions from Example 5 may be replaced by three triggers:

+IT1(R(x,y)) -INSERT(S(y)).
+DT1(S(y)) -R(x,y) -DELETE(R(x,y)) -DT1(S(y)) -/.
+DT1(S(y)).

Triggers may perform view updates identically as corrective actions above.
Our approach to the consistency of PROLOG databases is rather preliminary. In fact, we omitted the most

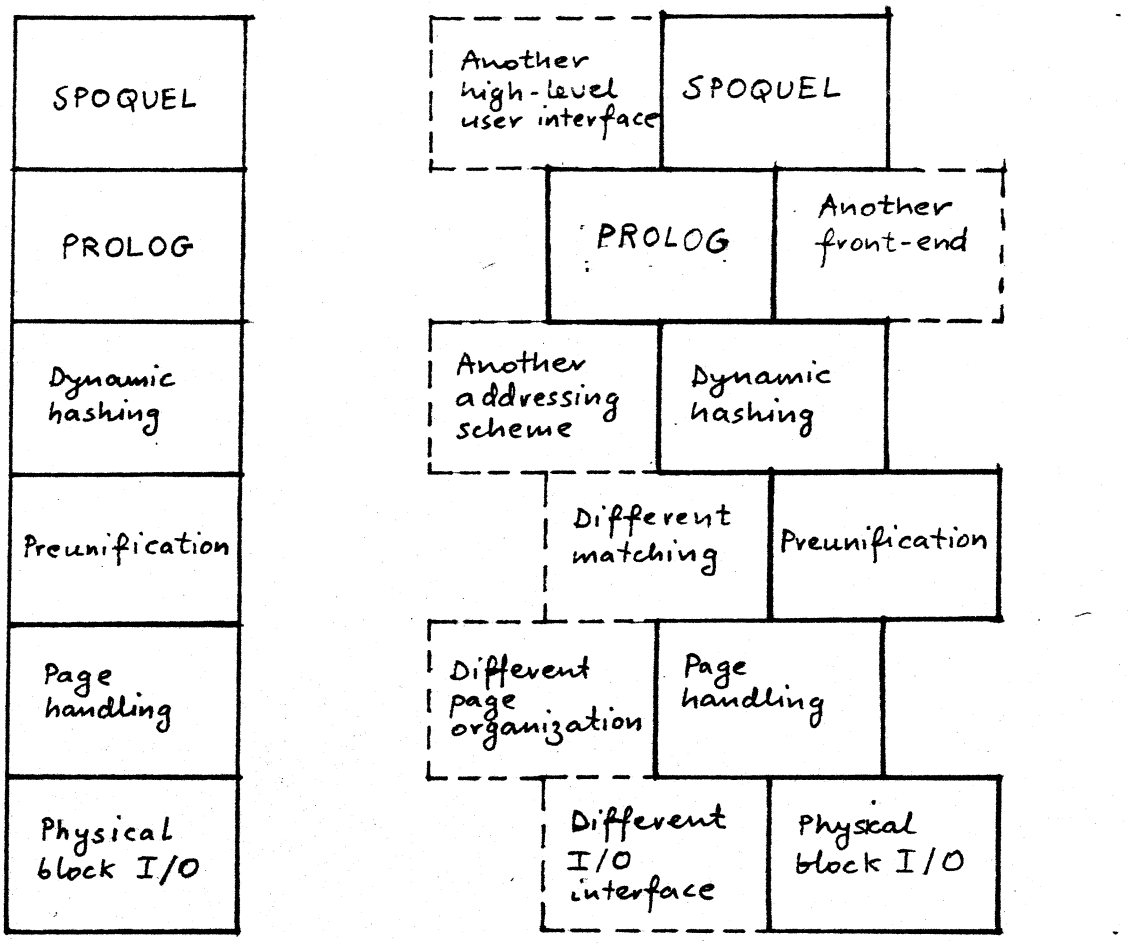
difficult problems of:

- supporting transactions (multiple actions) as consistency units (Eswaran 75)
- backing out the transactions violating database consistency
- efficient checking of arbitrary integrity assertions during or at the end of a transaction.

We do not think that the solution of the above problems is any easier with PROLOG than with conventional programming languages.

3.System architecture

System architecture may be described as an hierarchy of six levels. This paper discusses only the lowest four.



1.SPOQUEL is a SEQUEL-like query language described in (Kluzniak 83b). The interpreter of SPOQUEL is written in PROLOG.

2.The PROLOG interpreter (Kluzniak 83a) underwent only a minor change by incorporating new database primitives described in the next section.

3.Upon receiving a request from PROLOG, the addressing layer forwards the request to specific database pages. This layer performs also simple catalog management to keep track of definitions of relations.

4.Preunification filters out clauses retrieved by page handling layer and not matching the clause passed from PROLOG as the argument of the request. The variables are not instantiated, but only the matching clauses are returned. The unification is completed by the interpreter.

5.The page handling layer performs software paging and retrieves (inserts,deletes) tuples and auxiliary information from core buffers. The tuples are stored as variable-length and arbitrarily nested records. The page size is 2048 bytes.

6.Physical block I/O which supports paging is based on the file system of the underlying operating system.

Now consider the (relatively) easy-to-do alternatives marked by dashed lines.

The effects of their introduction would be limited to to the layer of their application.

1.Another high-level user interface, eg Query-By-Example, may be implemented in PROLOG analogously to SPOQUEL.

2.Another front-end would make possible to get into our system beside PROLOG. We actually had to develop

such a front-end for the purpose of testing.
3.If dynamic hashing does not turn out to perform satisfactorily in some applications, we may choose to replace it by another addressing scheme, eg multidimensional binary search trees. Such a possibility was outlined in (Chomicki 83).
4.Currently the preunification adheres to standard PROLOG semantics. However, it would be easy to adapt the preunification to somewhat different requirements, eg pattern-matching in text.
5.The variable-length records are flexible but may turn out to be inefficient, requiring an additional level of indirection. For conventional formatted databases, fixed length records should rather be considered.
6.It is also conceivable to bypass the file system and perform the I/O directly without protection and bookkeeping overhead.

On the whole, the interfaces between the layers are narrow, the layers are loosely coupled and besides there seems to be a high potential of asynchronous processing.

The PROLOG interpreter and the support system are written in CDL-2 (Koster 75), a highly modular language. CDL-2 gives the possibility of gluing high-level control and parameter-passing structures together with assembler macros. The support system consists of 3000 lines of source code. About 15% of code are written in MACRO-11 assembler. The system is being developed on the SM-4 minicomputer (functionally equivalent to PDP-11/40) under the RSX-11M operating system.

4. Database Support System.

In our database support system we have implemented extendible hashing scheme based on (Fagin 79) with an extension to partial-match retrieval along the lines proposed by (Lloyd 82). That method was chosen because of very good search performance which doesn't deterioriate for dynamic (growing and shrinking) files. This method is simple to implement, even with an extension to handle partial-match queries typical for PROLOG oriented databases.

This scheme has been adapted to handle incomplete information (Chomicki 83).

In the sequel we shall refer to PROLOG unit clauses as tuples.

4.1. Extendible hashing.

Extendible hashing is a method for handling dynamic files. There is a hash function h from the key space (domains of attributes) of the tuples to the set of bit strings of length k

h : K -> Bk

The details of the hash function will be discussed in section 4.2.

The file is structured into directory and database level.

The directory contains pointers to database pages and is characterized by a number, called the depth of the directory. The directory entries are indexed by all bit strings of the length d (d<=k) and d is the current depth of the directory. We called these bit strings (after (Lloyd 82)) the indexing strings.

When a directory entry is indexed by the string b1.....bd it means that the database page, pointed at by this entry, stores all the tuples for which h(K) starts with b1.....bd.

There are 2**d entries in the directory.

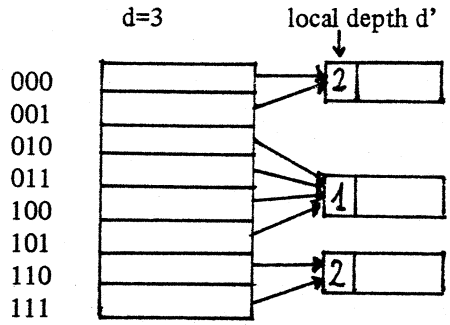


Fig 4.1

Database pages contain tuples. The order of tuples is immaterial. They are stored in structured compact form as variable-length arbitrarily-nested records. Each database page has a header which contains the local depth d' for this page. The local depth of a page is d' (d'<=d) iff there are 2**(d-d') entries in the directory, pointing at that page.

In Figure 4.1 the local depth of database page pointed at by the first directory entry is 2 and depth of the directory is 3. That means that this page contains all tuples for which h(K) agrees with 000 on the first 2 places. Thus the 001 entry also points at this database page.

Suppose that we want to insert a new tuple with a key K0. We calculate h(K0) and select its first d bits. Next we do a simple computation to find a coresponding entry in the directory. Following the pointer we find a database page on which a tuple should be placed. When this page is already full and d>d' then t splits into two database pages.

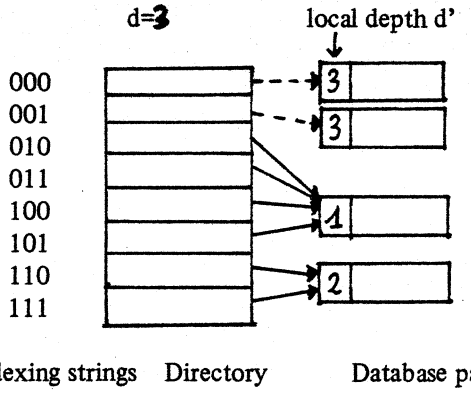


Fig 4.2

A new, initialy empty, database page is allocated and the tuples from the full page are hashed again. If the d'+1 th bit from h(K) of a tuple is 1 it is put on the new page. Otherwise it remains on the old one. The new tuple is treated identically. The local depths of both pages are set to d'+1.

If the database page is full and local depth is equal to the depth of directory (d=d') then the directory doubles in size as shown on the Figure 4.3 and the database page splits. The depth of the directory is increased by 1.

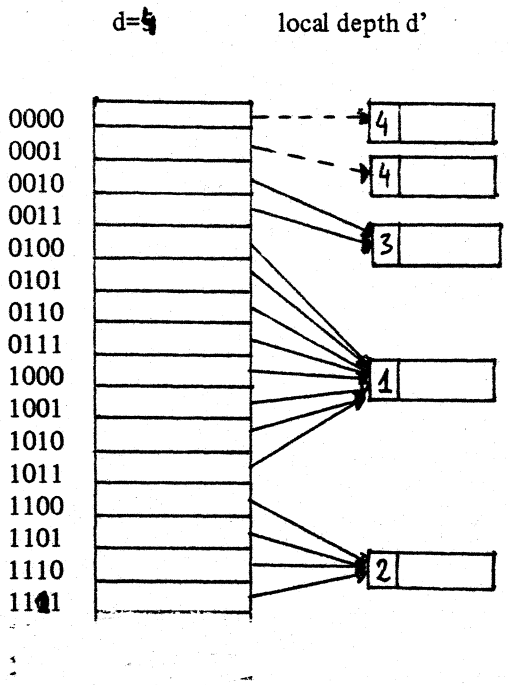


Fig 4.3

There is no need to access database pages during the doubling of the directory. When the directory exceeds one page, new directory page must be allocated during its doubling. However it does not happen very often.

In the case when the file is shrinking significantly after a large number of deletions and dictionary occupied many pages, it can be reduced twice in size.

4.2. Partial-match retrieval using extendible hashing.

The hash function h : A →Bk where A - set of all domains of relation's attributes is constructed. We are using one hash function (a kind of square hashing) for all attributes (but different hash functions can be constructed for each domain). The hash function uses as arguments the top level functor of clause attributes.

Let the value of the attribute ai in a query be vi for i=1,...,n. The final string which indexes a directory entry is constructed by selecting bits from each h(vi) according to a predefined choice vector (i1,i2,....,id) where d-depth of the directory. The m'th position in the indexing string will be filled by the first so far unused bit in the string h(vim).

Example 7.

When (4,1,4,2) is a choice vector then the indexing string contains
- first bit from the string h(v4),
- first bit from the string h(v1),
- second bit from the string h(v4),
- first bit from the string h(v2).

The methods of designing choice vectors are not considered here. The reader is referred to (Lloyd 80) and (Lloyd 82) for a discussion of this issue.

A partial-match query is a query in which an arbitrary subset of the attributes of the tuple is specified.

A query and a tuple match if all theirs specified attributes values are equal.

The result set of a query is the set of all pages where a matching tuples may reside.

When an incomplete query is considered and the choice vector indicates a bit from the unspecified attribute on the m'th place, then result set doubles, because both 0 and 1 should appear on the m'th place of all appriopriate indexing strings.

Example 8

For a relation T, choice vector (1,2,1) and the value of the hash function
for the second attribute h(GREEN)=010 the query
-T(x,GREEN,15) has the result set indexed by strings :
000, 001, 100, 101.

For a fully specified (complete) query only one page address is computed, so no more than two page accesses are necessary to find an answer.

4.3. Storage and retrieval of incomplete tuples.

The value of 'any' is introduced which is equivalent to PROLOG variable in the sense that 'any' matches every value in each domain and itself. When incomplete tuples appear, the set of tuples possibly matching a query grows exponentialy with the number of any-valued (unspecified) attributes.

Example 9

The query -S(TOKYO,1964) has four possible matchings :
+S(any,any), +S(any,1964), +S(TOKYO,any), +S(TOKYO,1964).

When an unspecified attribute provides no bits for the choice vector, the tuple is treated as complete.

We implemented a parametrized family of methods of the storage and retrieval of incomplete tuples.

Let m be the length of choice vector for the relation F and t an incomplete tuple to be inserted into F.
There are two extremal methods :
Method m (full replication)

- compute the result set of t and put a copy of t on each database page of this set. This method is time-optimal because the number of page accesses it requires is equal to the cardinality of the result set of a query.

However it gives a high storage redundancy - in the worst case 2**m.

Method 0 (no replication)

- put t on an arbitrary chosen page from the result set of t. This method is space-optimal, but search time can be unacceptable, requiring an access to each page, for we have no cues whether and where the incomplete tuples matching the query reside.

There is a family of intermediate methods numbered from 1 to m-1.

Method i

- tuples from each database page indexed by string b1...bi-1 bi bi+1....bm obtained by Method i-1 are specified in the following way :

if the i-th bit of the choice vector is given and equal ci then move the tuple to the database page indexed by the string b1.. b1.....bi-1 ci bi+1....bm,

otherwise put the tuple on both database pages indexed by strings b1....bi-1 0 bi+1.....bm and b1.....bi-1 1 bi+1.....bm.

In other words, to insert a tuple t, the set of indexing strings is constructed. First i-bits are selected in the way described in the previous section. Then to each of them m-i bits are appended and the copies of t are put on each indexed database page.

The parameter i, called any-depth, can be chosen by the user or be given by a system option.

It should be taken into account that the properties of Method i are changing during file evolution. As long as the depth of the directory is less qr equal than any-depth, full replication is performed.

When a choice vector is chosen then attributes which can assume the value of 'any' should rather not serve as a source of bits for the choice vector, because the result set for unspecified queries and consequently the number of accesses to database pages are growing.

4.4. Modyfing a file with incomplete tuples.

A partial ordering $\preceq$ is defined (interpreted as " t1 is no less precise then t2 ") among the tuples as follows :
t1 $\preceq$ t2 iff t1=(a1,.....an), t2=(b1,.....,b2) and ai=bi or bi=any for all i=1,...,n.

To insert a tuple we simple put it on the database page determined by an any-depth parameter in the way de cribed in previous sections .

The deletion from a file has a tuple t0 as an argument and is defined in two basic ways.
A. Delete tuple (delete no less precize) : delete all tuples t such that t $\preceq$ t0.
B. Delete this tuple (exact delete) : delete only those tuples t such that t=t0.

Example 10.

The request of deleting all tuples of relation R with first attribute equal to 1
may result in deleting

A. +R(1,YELLOW),....,+R(1,GREEN), +R(1,any)
B. +R(1,any).

The deletion of all tuples matching this tuple is not acceptable for it causes unintended loss of information. In the above example

+R(1,YELLOW),....., +R(1,GREEN), +R(1,any),
+R(any,YELLOW),....., +R(any,GREEN), +R(any,any).

Update is implemented as consecutive deletion and insertion in two variants.
1. Delete this tuple(t1),Insert(t2) - to update exactly one tuple.
2. Delete tuple(t1),Insert(t2) - to replace a set of tuples by one tuple.

4.4. Additional aspects of implementation.

Descriptions of all relations (PROLOG procedures) which are stored in the database are in a catalog.
For each relation the catalog contains :
- the unique identifier,
- the cardinality,
- the choice vector,
- the any-depth,
- the current depth of the directory,
- the address of the directory descriptor.

Each relation has its own directory. During the session the catalog resides in core and at the end of the session it is copied back to disk as the header of the database.

During query evaluation or modification preunification, which directly corresponds to unification in PROLOG, is performed on all levels. Only the binding of variables is left to the PROLOG interpreter.

There is a stack of queries which is used to handle of backtracking. Matching tuples are returned to the PROLOG interpreter one by one. The computation continues and, after backtracking, may return to one of the previous queries and request another matching tuple.

The stack for each query contains :

- indexing string which determines the address of last accessed database page;
- location of the last returned tuple on the database page;
- the pattern of indexing string (the string with 1 on the places fixed by specified attributes of query or tuple) which is used to find the next elements of a query result set;
- the local depth of database page during last matching.

A priority mechanism connected with buffer management is used for optimizing number of page accesses. The database page, on which more than one tuple matching a query reside, is kept in a buffer so long as it is possible.

However for a query which required many backtracking, in particular for a nested query , we cannot avoid many disk accesses for the same database page.

Another problem arises when, between two matchings for the same query (before backtracking), the database page on which we found last matching tuple splits. For example it can occur during the checking of integrity constraints for insertion, when a few additional tuples are stored. In this case the order of tuples is changed as an effect of a database page spliting, so the the next tuple is undefined.

Our solution is provisional. We retrieve all matching tuples from both old and new pages. It causes that some of the matching tuples are returned twice to the PROLOG interpreter like in the case of replication of incomplete tuples when the interpreter also receives duplicates and handles them. It seems that duplicate elimination would require quite a lot of additional data structures so we postponed it to the future development of the system.

4.6. Interface with PROLOG.

The interface between the PROLOG interpreter and the support system is very simple. The interpreter sends a request which is fulfilled and, if necessary, the matching clauses are returned one by one. The requests are treated by PROLOG like I/O commands.

The requests are :

Create relation(name, cardinality, any-depth) - adds a description of relation to the catalog and creates a new directory for it. The system creates a rest of a description. (The choice vector may be also a request parameter).

Drop relation(name) - deletes all tuples (if there are any), frees all database and directory pages and removes a relation from catalog.

Insert tuple(relation name, tuple address) - inserts a tuple to database.

Delete tuple(relation name, tuple address) and Delete this tuple(relation name, tuple address) - deletes tuples or tuple in the way described in section 4.4.

Delete all tuples(relation name) - deletes all tuples of an indicated relation.

Give first tuple(relation name, query pattern address, request number) - returns first tuple matching specified query pattern and pointer to the request stack entry which contains description of this request.

Give next tuple(relation name, query pattern address, request number) - returns next tuple matching specified query pattern (generally after backtracking). The pointer to request stack entry doesn't change.

The PROLOG interpreter and support system are working as synchronous processes.

5.Conclusions

Our proposals in section 2.3 and 2.4 demonstrated the conceptual conciseness of PROLOG in dealing with several database problems. The implementation of simple integrity assertions and triggers does not require any extensions of the language. The deductive capabilities of PROLOG are unquestionable. This all makes PROLOG an attractive programming language for an implementor of sophisticated user interfaces for databases. However,

using PROLOG in an actual database application requires further development of database management system mechanisms.

No form of recovery, concurrency control and protection is provided in our system, so it is certainly not a "full-fledged" database management system. It is nevertheless, to our knowledge, the first attempt to handle non-toy PROLOG databases. Other approaches: (Lloyd 82), (Kunifuji 82), (Warren 81) neglected the problems of: efficient secondary storage organization and access, the support for null values and views, and even the simplest integrity checking. We have proposed solutions for the above problems and incorporated the solutions into an actual system. The experience with this system will certainly give further arguments for (or against) the use of PROLOG in the database area.

6. Acknowledgements

We are greatly indebted to Feliks Kluzniak and Stanislaw Szpakowicz for the constant inspiration and help in the development of our system.

7. References.

(ANSI 75) ANSI/X3/SPARC Interim Report, FDT Bulletin of ACM-SIGMOD, 7:2, 1975.

(Blaustein 81) Blaustein,B.T., Enforcing database assertions : techniques and applications. Ph.D. thesis Harvard University, 1981.

(Bowen 81) Bowen,K.A. and Kowalski,R.A., Amalgamating language and metalanguage in logic programming. School of Computer and Information Science, Syracuse University, 4/81, June 1981.

(Chomicki 83) Chomicki,J., Implementing null values. (Submitted for publication).

(Clark 79) Clark,K.L., Negation as failure in : Logic and data bases, Eds. Gallaire,H. and Minker,J. Plenum Press 1979.

(Codd 79) Codd,E.F. Extending the relational data base model to capture more meaning. ACM Transactions on Database Systems, 4,4, 1979.

(Dayal 82) Dayal,U. and Bernstein,P.A., On the correct translation of update operations on relational views. ACM Transactions on Database Systems, 7,3, 1982.

(Eswaran 75) Eswaran,K.P. and Chamberlin,D.D., Functional specifications of a subsystem for database integrity. Proceedings , 1st Int'l Conf.on Very Large Data Bases, 1975.

(Eswaran 76) Eswaran,K.P., Specifications, implementations and interactions of a trigger subsystem in an integreted database system. IBM RJ 1820, August 1976.

(Fagin 79) Fagin,R. at all, Extendible hashing - a fast access method for dynamic files. ACM Transactions on Database Systems, 4,3, 1979.

(Fernandez 81) Fernandez,E.B. and Summers,R.C. and Wood,C., Database security and integrity. Addison-Wesley 1981.

(Kluzniak 83a) Kluzniak,F., PROLOG for SM-4. Technical documentation, Institute of Informatics, Warsaw University 1983.

(Kluzniak 83b) Kluzniak,F. and Szpakowicz,S., SPOQUEL - a Simple PROLOG-Oriented Query Language. Institute of Informatics, Warsaw University (in preparation).

(Koster 74) Koster,C.H.A., Using the CDL compiler-compiler. In : Compiler construction, an advanced course. Eds. Bauer,F.J. and Eickel,J. Lectures notes in computer science 21, Springer-Verlag 1974.

(Kunifuji 82) Kunifuji,S. and Yokota,A., PROLOG and relational databases for fifth generation computer systems. In : Preprints,Workshop on Logical Bases for Databases, Toulouse, France, Dec.1982.

(Lloyd 80) Lloyd,J.W., Optimal partial-match retrieval. BIT 20,1980.

(Lloyd 82) Lloyd,J.W., An introduction to deductive database systems. TR 81/3, Dept.of Computer Science, Univ.of Melbourne, revised 1982.

(Naish 83) Naish,L., Introduction to MU-PROLOG. TR 82/2, Dept.of Computer Science, Univ.of Melbourne, revised 1983.

(Nicolas 82) Nicolas,J.M., Logic for improving integrity checking in relational databases. Acta Informatica 18,1982.

(Paolini 82) Paolini,P. and Zicari,R., Properties of views and their implementation. In : Preprints, Workshop on Logical Bases for Databases, Toulouse, France, Dec.1982.

(Roussel 75) Roussel,Ph., PROLOG, manuel de reference et d'utilisation. Groupe d'Intelligence Artificielle, Universite d'Aix-Marseille II, 1975.

(Siklossy 82) Siklossy, L., Updating views : a constructive approach. In : Preprints, Workshop on Logical Bases for Databases, Toulouse, France, Dec.1982.

(Ullman 80) Ullman,J.D., Principles of Database Systems. Computer Science Press, 1980.

(Vassiliou 80) Vassiliou,Y., A formal treatment of imperfect information in database management. TR CSRG-123, Univ.of Toronto, Nov.1980.

(Warren 81) Warren, D.H.D., Efficient processing of interactive relational database queries expressed in logic. Proc.7th.Int'l Conf.on Very Large Data Bases, Cannes, France, Sept.1981.

(Zaniolo 82) Zaniolo,C., Database relations with null values. Proc.ACM Symp.on Principles of Database Systems, 1982.

SECURITY AND INTEGRITY IN LOGIC DATA BASES USING QUERY-BY-EXAMPLE

M H WILLIAMS, J C NEVES and S O ANDERSON

Department of Computer Science

Heriot-Watt University

Edinburgh

Scotland

KEYWORDS: Security, Integrity, Query-by-Example, Prolog, Logic data base.

Abstract

Security and integrity are two important and inter-related aspects of data base systems, and data base management languages must make provision for the specification and enforcement of such constraints. In the case of the data base language Query-by-Example a style for handling certain types of security and integrity constraints has been developed by Zloof.

An alternative approach to integrity in QBE is presented here which is based on the idea of consistency of the data in the data base. This approach allows for a more general type of constraint which includes the handling of functional, multivalued and embedded-multivalued dependencies, as well as the more conventional and simpler type of integrity constraints in a uniform manner.

Both security and integrity constraints have been implemented in Prolog as part of a logic data base.

1. INTRODUCTION

One of the important functions of any data base management system is to preserve the integrity of any data stored within the data base by ensuring that it is consistent with the prescribed properties of such data (integrity constraints). Integrity constraints can be classified into three types (Ullman [1], Nicolas and Yazdanian [2]) :

(a) Value-based constraints. These are conditions which the values of the domain elements must satisfy. They are usually restrictions on the range of values which a field can assume or are concerned with non-structural relationships amongst various fields. For example in the set of relations given in Appendix 1 one might wish to impose restrictions such as:

(i) The weight of a part is always less than 100 units (simple restriction on range).

(ii) An entry may only appear in the supplier_parts table if an entry for the supplier concerned exists in the supplier table (existence check).

(iii) Any supplier from Vienna or Athens must have a status which is at least 20 (non-structural relationship), etc.

(b) Structural or "Value-oblivious" constraints. These are restrictions concerned not with the value in any

particular field of a tuple but with whether certain fields of one tuple match those of another. Three specific types of structural constraints are addressed in this paper:

(i) Functional Dependencies. If X and Y are two sets of attributes from some relation scheme, then X functionally determines Y (or Y functionally depends on X), written "X -> Y", if any pair of tuples which agree in the components for all attributes in set X must likewise agree in all components corresponding to attributes in set Y.

Examples of functional dependencies in the set of relations in Appendix 1 include:

> sno -> sname (corresponding to each supplier number is a unique name),
>
> sno, pno -> qty (corresponding to each supplier/part number combination is associated an unique quantity),

and so on. It has been shown [3] that any set of functional dependencies can be transformed to an equivalent set in which all functional dependencies have the form "X -> Y" where Y is a singleton set.

(ii) Multivalued Dependencies. If X and Y are two sets of attributes from some relation scheme then X multidetermines Y (or there is a multivalued dependency of Y on X), written "X ->-> Y", if corresponding to a given set of values for the attributes of X there is a set of zero or more associated values for the attributes of Y, and this set of Y-values is independent of the values of any attributes

not contained in X U Y.

An example of a multivalued dependency taken from the relation scheme in Appendix 2 (taken from Ullman [1]) is:

course ->-> period, room, teacher

that is, associated with each "course" is a set of "period-room-teacher" triples which does not depend on any other attributes. For example, given the pair of tuples:

cs2a	3	601	jones j	adams a	42
cs2a	5	302	smith t	zebedee e	67

one would expect to be able to exchange (3, 601, jones j) with (5, 302, smith t) and obtain two valid tuples, viz:

cs2a	5	302	smith t	adams a	42
cs2a	3	601	jones j	zebedee e	67

However, it is not possible to exchange one or two fields of the triple without exchanging all of them, eg:

cs2a	5	601	smith t	adams a	42

is not in the data base since "course ->-> room" does not hold.

(iii) Embedded Multivalued Dependencies. These are multivalued dependencies which do not apply in the full set of data but which become applicable when the data set is reduced by projection. Formally, given a relation scheme R, an embedded multivalued dependency is one which holds only when any relation r in R is projected onto some subset X [

R. For example, in the relation scheme presented in Appendix 2, the multivalued dependency "course ->-> prerequisite" does not hold since tuples such as:

cs2a zebedee e cs1b 1978

are not present in the data base. However, if the data in progresstable is projected onto the subset {course, student, prerequisite} giving:

cs2a	adams a	cs1a
cs2a	adams a	cs1b
cs2a	zebedee e	cs1a
cs2a	zebedee e	cs1b

then "course ->-> prerequisite" does hold, as does "course ->-> student".

(c) Transition constraints. These are restrictions on the way in which the data base may change; or, more specifically, the relationship between the states of the data base before and after any change is made. They include restrictions on the way in which:

(i) Values in a single field may change, e.g. values such as age or salary may only increase, marital status may only change in a particular way, etc.

(ii) Values in a set of fields (possibly in different relations) may change, e.g. the amount of special low-interest-rate loan may be increased only if the grade of the employee is above a certain level, etc.

Security, on the other hand, is concerned with who may access what information in the data base and what operations may be performed. The distinction between security and integrity constraints is not always clear as will be seen in later sections.

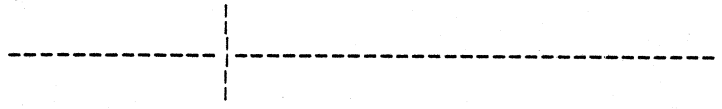
Zloof [4] has developed mechanisms for handling security and integrity constraints within the data base language Query-by-Example (QBE). The approach used for handling integrity constraints is a trivial extension of the concept of transition constraints in which constraints may be placed on insert, delete and update operations as well as on print operations. The problem with such an approach is that it is not possible to make any general statements about the data in the data base without a detailed history of the data base.

The object of this paper is to present a slightly different approach which includes all three types of constraints, and which does lend itself to statements about the properties of data in the data base.

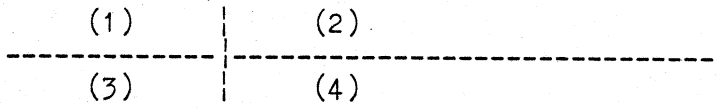
The following section gives a brief introduction to Query-by-Example, while section 3 looks briefly at the specification of security constraints (a slight variation from Zloof's approach). The remainder of the paper is devoted to the integrity constraints and implementation details.

2. QUERY-BY-EXAMPLE - THE BASIC LANGUAGE

Query-by-Example [5] is a two-dimensional language which is designed for use at a terminal and makes use of a special-purpose screen editor to compose queries. On striking a particular key, the user is presented with the skeleton of a table as follows:



The four areas delimited by this skeleton are:



(1) Table name field,

(2) Column name field,

(3) Tuple command field, and

(4) Tuple entry field.

Using the screen editor the user may position the cursor in any of these four areas in order to insert a command and/or a variable or constant element. The formulation of queries is achieved by setting up tuples containing variables, constants and conditions. An attribute which is to be displayed is indicated to the system by typing "p.", followed possibly by a variable name and possibly by a condition, in the column corresponding to that attribute. In our implementation lower-case letters have

been used in place of upper-case letters for the basic operations.

For example, to print the status of a particular supplier, say "clark", given the data base of Appendix 1, the user may enter the table name "suppliers" in the table name field, viz (the parts which the user might enter are underlined "___"):

suppliers	
----------	----------

Since the relation already exists in the data base the column headings (attributes of suppliers) can be generated by the system, i.e.:

suppliers	sno	sname	status	city
----------	----------			

One can now enter "clark" in the sname field and "p.X" in the status field as follows:

suppliers	sno	sname	status	city
----------	----------			
		clark	p.X	

Any character sequence beginning with a lower case letter, such as "clark", is taken to be a constant representing a specific value, while one beginning with an upper case letter or an underline symbol "_" is taken to be

a variable. Thus this is interpreted as a request to print the status of any supplier whose name is "clark".

Similarly to print the details of any supplier whose status exceeds 10, one may enter:

suppliers	sno	sname	status	city
	p.X	p.Y	p.A::A>10	p.C

or one may write the command "p." in the tuple command field as follows:

suppliers	sno	sname	status	city
p.	X	Y	A::A>10	C

where the infix operator "::" is used as a syntactic aid and is to be read as "such that".

A query may require more than one relation in which case appearances of the same variable name in different parts of a query represent the same value. For example, to display the names of all suppliers who supply parts which are red, one may enter:

parts	pno	pname	colour	weight
	X		red	

supplier_parts	sno	pno	qty
	Y	X	

suppliers	sno	sname	status	city
	Y	p.Z		

Complex conditions are handled by use of a separate condition box. For example, suppose that one wishes to display the names of all suppliers for whom the quantity of part number 2 lies between 100 and 300. One may enter:

suppliers	sno	sname	status	city
	X	p.Y		

supplier_parts	sno	pno	qty
	X	2	Z

CONDITIONS
Z>99 and Z<301

Besides the query operator "p." there are three other

operators: "i." (Insert), "d." (Delete) and "u." (Update). As an illustration of the use of "i.", consider the addition of a new part tuple to the relation parts:

parts	pno	pname	colour	weight
i.	7	washer	red	10

3. SECURITY IN QUERY-BY-EXAMPLE

Security constraints take the form of an authorization for a user to perform certain operations on a relation. For example, if one wishes to permit a user John to perform print, update and insert operations on the relation suppliers, this may be specified as follows:

suppliers	sno	sname	status	city
i.autr(p.,u.,i.).john	A	B	C	D

where once again lower case letters have been used and the final "i." omitted [4].

The presence of a variable in each field of the relation indicates that John has access to that field. If the variable C had been omitted and the status field left blank, this would indicate that John does not have access to the status field. Just as in other QBE statements, one may add conditions to these variables or link them to fields in other relations.

A more complex example which illustrates this imposes the constraint that John may only read details from the supplier_parts relation if the status of the supplier is less than 30 or the supplier comes from Paris. This is specified as follows:

supplier_parts	sno	pno	qty
i.autr(p.).john	A	B	C

suppliers	sno	sname	status	city
	A		E	F

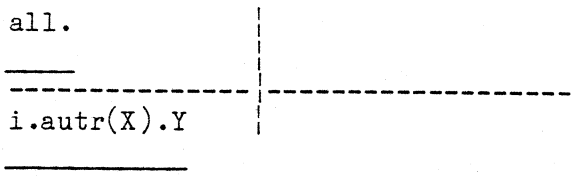
CONDITIONS
E<30 or F=paris

In each case the entry in the tuple-command-field has the form:

i.autr(<access rights lists>).<user>

The <access rights list> is a list of one or more of the four rights "p.", "i.", "u." or "d." while <user> is the name of the user to whom access is to be granted. In generalizing these two items, following the philosophy of QBE, variables may be used. Similarly if the keyword "all." is used in the table-name-field it will refer to all

relations. Thus the constraint:



will allow any user to perform any operation on any relation.

4. REALIZATION IN PROLOG

In our initial implementation of QBE in Prolog [6], each QBE request (insertion, deletion, update, print, constraints) was translated directly into Prolog and applied to the data base. However, when we changed our approach to integrity constraints and adopted the approach which will be described in the next section, a different implementation strategy was called for.

In the current system (which runs both on a PDP 11/34 and a DEC 10 machine), each QBE request is translated into a clause in a meta-language which is then interpreted using the remainder of the data base.

The following notation is used to express object-level knowledge in the meta-language:

(1) A rule clause is represented as:

p <- [q1, q2, ..., qn, {s}].

which stands for p :- q1,q2, ...,qn.
while the string s in braces {} is used
to store information for recreating the

original QBE request.

(2) A goal clause is represented by:

<- [q1,q2,...,qn].

which stands for ?- q1,q2,...,qn.

(3) A fact or assertion is represented as:

p.

which stands for p.

The usual interpretations are to be understood for rules, goals and assertions [7]. The use of the meta-language at the object-level has the great advantage of allowing one to use clauses and predicates as terms.

5. EXTENSION TO HANDLE INTEGRITY CONSTRAINTS

The general philosophy behind the approach described here is that any constraint which is currently operative must apply to all data in the data base. Thus whenever a new constraint is defined, it is immediately checked against the data in the data base. If any of the data does not satisfy the constraint, the exceptions are reported and the user is given the opportunity of either updating the data or revising the constraint. If all the data does satisfy the new constraint, it is stored and used to check all insertions and update operations conducted in the future. Three new operators are introduced for this purpose:

ic. - insert a new constraint

dc. - delete an existing constraint

pc. - print constraints

The form of a constraint definition is similar to that of a query. As a simple example, consider the insertion of the constraint that the value in the quantity field of each supplier_parts tuple should be greater than zero. To do this one may enter:

supplier_parts	sno	pno	qty
ic.			X::X>0

or one may use the condition box as follows:

supplier_parts	sno	pno	qty
ic.			X

CONDITIONS
X > 0

which is translated by the system to yield:

```
supplier_parts(_, _, X) <-
      [ X>0,
        {X>0}
      ].
```

To ensure that a tuple may only exist in the

supplier_parts relation if a tuple for the supplier concerned exists in the suppliers relation, one may have:

supplier_parts	sno	pno	qty
ic.	X		

suppliers	sno	sname	status	city
	X			

which is translated by the system to yield:

```
supplier_parts(X, _, _) <-
      [ suppliers(X, _, _, _),
        {}
      ].
```

A more complicated value-based constraint is the restriction that any supplier from Vienna or Athens must have a status which is at least 20. To specify this, one has:

suppliers	sno	sname	status	city
ic.			X	Y

CONDITIONS
(Y = vienna or Y = athens) implies (X >= 20)

which is translated by the system to yield:

```
suppliers(_, _, X, Y) <-
   [ not (Y=vienna or X>=20) and
     not (Y=athens or X>=20),
     {(Y=vienna or Y=athens) implies (X>=20)}
   ].
```

Functional dependencies are specified in the condition box using the format:

<var> -> <var>

or (<varlist>) -> <var>

For example, in the parts relation, suppose that "pno -> weight". This can be specified as a constraint as follows:

parts	pno	pname	colour	weight
ic.	X			Y

CONDITIONS
X -> Y

which is translated as follows:

```
parts(X, _, _, Y) <-
      [ parts(X, _, _, U),
        Y=U,
        {1->4}
      ].
```

This can be read as:

```
for all X, A, B, Y:
if there exists R, S, U such that
if parts(X, R, S, U) and Y=U are true
then parts(X, A, B, Y) is true.
```

When this command is given, the data base will be checked immediately to ensure that the data already present satisfies this condition. Provided it does, the constraint will be added to the data base. Thereafter whenever the user inserts or updates a tuple in the parts relation it attempts to deduce "weight" from "pno" and fill it in automatically for the user.

Multivalued dependencies are specified in a similar way using the format:

<X> ->-> <Y>

where <X> and <Y> each stand for either a single variable or a variable list enclosed in parentheses. Thus in the example from Appendix 2 one might express the constraint:

timetable	course	period	room	teacher	student	mark
ic.	W	X	Y	Z		

CONDITIONS
W ->-> (X, Y, Z)

which is formalized as follows:

```
timetable(W, X, Y, Z, R, S) <-
      [ timetable(W, A, B, C, M, N),
        timetable(W, A, B, C, R, S),
        timetable(W, X, Y, Z, M, N),
        {1->->(2,3,4)}
      ].
```

Once again when this command is given the data base is checked for any violations. If violations arise they are reported, if not the constraint is added to the data base. Thereafter whenever an insertion or update operation causes this constraint to be invoked, the system generates (and displays) the full set of tuples which need to be added to the data base in order to maintain consistency. If the user is content with the set of tuples generated, the system adds the full set to the data base, otherwise the insertion/update operation is abandoned.

Embedded multivalued dependencies are specified using the format:

<X> ->-> <Y>/<Z>

where <X>, <Y> and <Z> each stand for either a single variable or a variable list in parentheses. This is interpreted as X multidetermines Y if the set of attributes Z is removed. For example, to express the fact that "course ->-> prerequisite" if the relation "progresstable" in Appendix 2 is projected onto the subset {course, student, prerequisite}, one may enter:

progresstable	course	student	prerequisite	year
ic.	X		Y	Z

CONDITIONS
X ->-> Y/Z

which is translated by the system to yield:

```
progresstable(X, A, Y, Z) <-
      [ progresstable(X, B, C, E),
        progresstable(X, A, C, R),
        progresstable(X, B, Y, S),
        {1->->3/4}
      ].
```

When this command is given, the data base is checked for consistency. If violations arise the user is prompted to correct them or abort the constraint. Once the constraint is added to the data base, any further insertions or update operations are checked against the constraint and where required the system will generate the full set of tuples needed to fulfil any particular operation, prompting the user for the additional information (year) required to complete each tuple.

Transition constraints, which are concerned with the way in which values in the data base may change, are expressed using a pair of entries for the relation in question. The field in this relation which is to be controlled, will be represented by two different variables -

the one occurring in the line with the ic command in the tuple-command-field represents the new value of the variable, the other the old value.

For example, suppose that one wishes to place a constraint on the status of a supplier whereby it can only increase, one might enter:

suppliers	sno	sname	status	city
ic.	N		X	
	N		Y	

CONDITIONS
X>=Y

which is translated by the system to yield:

```
suppliers(N, _, X, _) <-
     [ suppliers(N, _, Y, _),
       X>=Y,
       {X>=Y}
     ].
```

Similarly one might impose a constraint on the age or salary of an employee whereby the values of these fields for a particular employee can only increase. In the case of marital status the only permissible transitions may be:

single	--->	married
married	--->	divorced or widowed
divorced	--->	married
widowed	--->	married

which may be specified as follows:

employee	empno	ename	salary	status	grade
ic.	N			X	
	N			single	

CONDITIONS
X=married or X=single

employee	empno	ename	salary	status	grade
ic.	N			X	
	N			married	

CONDITIONS
X=married or X=divorced or X=widowed

and so on. This is translated by the system to yield:

```
employee(N, _, _, X, _) <-
      [ employee(N, _, _, single, _),
        X=married or X=single,
        {X=married or X=single}
      ].

employee(N, _, _, X, _) <-
      [ employee(N, _, _, married, _),
        X=married or X=divorced or X=widowed,
        {X=married or X=divorced or X=widowed}
      ].

...
...
...
```

Alternatively the four constraints may be combined into a single one using two variables.

As an example of a more complex form of constraint, consider the restriction that the value of a loan may only increase (or decrease) if the grade of the employee is greater than 5. This might be specified as follows:

loantable	empno	loan
ic.	X	NL
	X	OL

employee	empno	ename	salary	status	grade
	X				Y

CONDITIONS
(Y<=5) implies (NL=OL)

This is translated by the system to yield:

```
loantable(X, NL) <-
      [ loantable(X, OL),
        employee(X, _, _, _, Y),
        not Y<=5 or NL=OL,
        {(Y<=5) implies (NL=OL)}
      ].
```

The complete syntax of these constraints is given in Appendix 3.

6. OVERLAP OF INTEGRITY AND SECURITY CONSTRAINTS

The transition constraints discussed in the previous section deal only with the way in which data in the data base may change (i.e. be updated). It does not cater for transitions involving insertion or deletion.

Thus suppose one wishes to impose the constraint that a loan may only be granted to an employee with grade between 5 and 8, but once an employee has been granted a loan, if his grade changes to a value outside the range 5-8, he will not lose his existing loan. This type of constraint is not a simple property of the data (i.e. one cannot conclude that any employee who has a loan, must have a grade in the range 5 to 8). However, it can be handled using a security constraint, eg.

loantable	empno	loan
i.autr(i.).X	A	B

employee	empno	ename	salary	status	grade
	A				C

CONDITIONS
(C>=5) and (C<=8)

Likewise the example considered by Nicolas and Yazdanian [2] in which a constraint needs to be placed on the system to prevent employees whose income is less than some value (say 5000) from being deleted, can be treated as follows:

employee	empno	ename	salary	status	grade
i.autr(d.).X	A	B	C	D	E
	—	—	—	—	—

CONDITIONS
C>=5000

7. CONCLUSIONS

The specification and enforcement of integrity constraints in a data base system is essential in order to guarantee the consistency of data within the data base. The role of security constraints is to control the types of operations which individual users may perform on the data base. The two types of constraints overlap to some extent.

This paper presents an integrated approach for specifying generalized integrity and security constraints within the data base management language Query-by-Example.

The important aspects of this approach are:

(a) It caters for all three types of integrity constraints in a generalized and consistent manner.

(b) It treats integrity constraints as properties of the data applying to all data in the data base, rather than as properties of particular operations (as proposed by Zloof [4]).

(c) It ensures that the user is aware of the

implications of any operation producing changes in the data base which affect fields involved in multivalued or embedded-multivalued dependencies.

ACKNOWLEDGEMENTS

The work of one of the authors, J C NEVES, was supported by the Calouste Gulbenkian Foundation under grant 14/82 and by an ORS award from the Committee of Vice-Chancellors and Principals of the Universities of the United Kingdom.

J C NEVES is on leave from Minho University, Largo do Paco, 4700, BRAGA, PORTUGAL.

8. REFERENCES

[1] J. D. Ullman, Principles of Database Systems, London: Pitman, 1980.

[2] J. M. Nicolas and K. Yazdanian, Integrity Checking in Deductive Data Bases, in: Logic and Data Bases, Plenum Press, 1978, 325-344.

[3] W. W. Armstrong, Dependency Structures of Database Relationships. Proc. IFIP 74, 1974, 580-583.

[4] M. M. Zloof, Security and Integrity within the Query-by-Example Database Management Language, IBM RC6982, Yorktown Heights, N. Y., 1978.

[5] M. M. Zloof, Query-by-Example: A Data Base Language, IBM Systems J., Vol. 16, No. 4, 1977, 324-343.

[6] J. C. Neves, S. O. Anderson and M. H. Williams, A Prolog Implementation of Query-by-Example, in: Proceedings of the 7th International Computing Symposium, March 22-24, 1983, Nurnberg, Germany.

[7] W. F. Clocksin, and C. S. Mellish, Programming in Prolog, Springer-Verlag, 1981.

[8] K. Bowen, and R. A. Kowalski, Amalgamating Object Language and Metalanguage in Logic Programming. To appear in Logic Programming (K. L. Clark and S. -A. Tarnlund Eds.)

Academic Press,1982.

[9] M. H. Williams, A Flexible Notation for Syntatic Definitions, ACM Trans. on Prog. Lang. and Syst., Vol. 4, No. 1, 1982, 113-119.

Appendix 1: A simple business data base

Consider a simple business data base which contains:

(i) A relation "parts" with attributes (columns): pno, pname, colour and weight.

(ii) A relation "suppliers" with attributes: sno, sname, status and city.

(iii) A relation "supplier_parts" with attributes: sno, pno and qty.

(iv) A relation "employee" with attributes: empno, ename, salary, status and grade.

(v) A relation "loantable" with attributes: empno and loan.

Suppose that the current content of each relation is:

parts	pno	pname	colour	weight
	1	nut	red	12
	2	bolt	green	17
	3	screw	blue	17
	4	screw	red	14
	5	cam	blue	12
	6	cog	red	19

Table 1.1 - The parts relation

suppliers	sno	sname	status	city
	1	smith	20	vienna
	2	jones	10	paris
	3	blake	30	paris
	4	clark	20	vienna
	5	adams	30	athens

Table 1.2 - The suppliers relation

supplier_parts	sno	pno	qty
	1	1	300
	1	2	200
	1	3	400
	1	4	200
	1	5	100
	1	6	100
	2	1	300
	2	2	400
	3	2	200
	4	2	200
	4	4	300
	4	5	400

Table 1.3 - The supplier_parts relation

employee	empno	ename	salary	status	grade
	12	morley	6500	married	10
	7	warren	7135	single	7
	15	exner	4475	single	4
	17	berry	5345	married	12
	5	john	6725	widowed	9

Table 1.4 - The employee relation

loantable	empno	loan
	7	570
	17	1500

Table 1.5 - The loantable relatio

Appendix 2: A simple departmental data base

Consider a simple departmental data base which contains:

(i) A relation "timetable" with attributes: course, period, room, teacher, student, grade.

(ii) A relation "progresstable" with attributes: course, student, prerequisite, year.

Suppose that the current content of the data base is:

timetable	course	period	room	teacher	student	grade
	cs2a	3	601	jones j	adams a	42
	cs2a	5	302	smith t	zebedee e	67

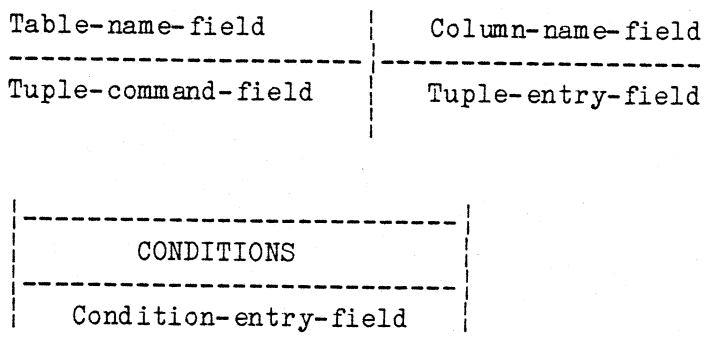
Table 2.1 - The timetable relation

progresstable	course	student	prerequisite	year
	cs2a	adams a	cs1a	1978
	cs2a	adams a	cs1b	1979
	cs2a	zebedee e	cs1a	1978
	cs2a	zebedee e	cs1b	1979

Table 2.2 - The progresstable relation

Appendix 3: Concrete syntax of the data base query language Extended-Query-by-Example

The basic Extended-Query-by-Example (EQBE) format is as follows:



where the syntax of each of these components is defined as:

```
table-name-field ::= ("i." | "u.") string-constant |
                     ["p." | "d."] [string-constant] |
                     "all."
column-name-field ::= ["p."] [string-constant]
tuple-entry-field ::= ["p."] [example-element
                      ["::" relation] | p-relation]
                      | string-constant | integer
authorization ::= "autr" ["(" access-rights-list ")"] "."
                  user-list
access-rights-list ::= access-right ("," access-right)* |
                       example-element
access-right ::= "p." | "i." | "d." | "u."
user-list ::= list | example-element | string-constant
list ::= "(" string-constant ("," string-constant)*")"
```

```
tuple-command-field ::= ["ic." | "dc." | "pc." |
                         ("i." | "d." | "u." | "p.")
                         [authorization]]
condition-entry-field ::= functional-dependency
                        | multivalued-dependency
                        | embedded-multivalued-dependency
                        | boolean-expression
functional-dependency ::= set "->" example-element
multivalued-dependency ::= set "->->" set
embedded-multivalued-dependency ::= set "->->" set "/" set
set ::= "(" example-element ("," example-element)* ")"
       | example-element
boolean-expression ::= boolean-secondary
                       ("implies" boolean-secondary)*
boolean-secondary ::= boolean-term ("or" boolean-term)*
boolean-term ::= boolean-factor ("and" boolean-factor)*
boolean-factor ::= ["not"] boolean-primary
boolean-primary ::= boolean-constant | relation
                  | "(" boolean-expression ")"
boolean-constant ::= "true" | "false"
relation ::= numeric-exp relational-op numeric-exp
           | string-exp relational-op string-exp
p-relation ::= relational-op (numeric-exp | string-exp)
numeric-exp ::= [add-op] numeric-term
                (add-op numeric-term)*
numeric-term ::= factor (multiply-op factor)*
factor ::= [function-designator] numeric-variable
```

```
            ¦ numeric-constant ¦ "(" numeric-exp ")"
multiply-op ::= "*" ¦ "/"
add-op ::= "+" ¦ "-"
function-designator ::= "max." ¦ "min." ¦ "ave."
                        ¦ "cnt." ¦ "sum."
string-exp ::= string-primary ("+" string-primary)*
string-primary ::= string-variable ¦ string-constant
integer ::= digit +
string-constant ::= (""""non-quote-character*"""")+ ¦
                    lower-case-letter letter-or-digit*
string-variable ::= example-element
numeric-variable ::= example-element
example-element ::= capital-letter letter-or-digit*
                    ¦ underscore letter-or-digit*
letter-or-digit ::= lower-case-letter ¦ digit
capital-letter ::= "A" ¦ "B" ¦ "C" ¦ "D" ¦ "E" ¦ "F"
                   ¦ "G" ¦ "H" ¦ "I" ¦ "J" ¦ "K" ¦ "L"
                   ¦ "M" ¦ "N" ¦ "O" ¦ "P" ¦ "Q" ¦ "R"
                   ¦ "S" ¦ "T" ¦ "U" ¦ "V" ¦ "W" ¦ "X"
                   ¦ "Y" ¦ "Z"
lower-case-letter ::= "a" ¦ "b" ¦ "c" ¦ "d" ¦ "e" ¦ "f"
                      ¦ "g" ¦ "h" ¦ "i" ¦ "j" ¦ "k" ¦ "l"
                      ¦ "m" ¦ "n" ¦ "o" ¦ "p" ¦ "q" ¦ "r"
                      ¦ "s" ¦ "t" ¦ "u" ¦ "v" ¦ "w" ¦ "x"
                      ¦ "y" ¦ "z"
digit ::= "0" ¦ "1" ¦ "2" ¦ "3" ¦ "4" ¦ "5"
          ¦ "6" ¦ "7" ¦ "8" ¦ "9"
```

where the notation used is that given by Williams [9].

TOWARDS A CO-OPERATIVE DATA BASE MANAGEMENT SYSTEM

J C NEVES and M H WILLIAMS

DEPARTMENT OF COMPUTER SCIENCE

HERIOT-WATT UNIVERSITY

EDINBURGH

SCOTLAND

KEYWORDS: Logic programming, Horn clauses, Prolog, Logic data base, Query-by-Example, Incomplete queries, Co-operativeness.

Abstract

A desirable feature of any high-level data base query system is that it should be user-friendly. This should extend beyond the provision of a query syntax which is easy to use, to some attempt at intelligent helpfulness or co-operativeness. In particular additional knowledge about the structure of the data in a data base or the incomplete data contained in a query may be used to benefit the user. In this respect, despite its simplicity and ease of use, the data base management language Query-by-Example is relatively inflexible.

This paper looks at several ways in which the co-operativeness of Query-by-Example can be improved. These are concerned with incomplete queries (i.e. queries in which certain information has been omitted), incomplete updates and queries which fail as a result possibly of misconceptions on the part of the user. Consideration is also given to how these are implemented in Prolog.

1. Introduction

The application of first order logic and resolution based theorem proving to machine intelligence problems started during the early 1970's. Recently, logic programming has received a considerable boost due to its choice as the basis of the core programming languages for the Japanese Fifth Generation Computer Systems [1].

Prolog [2] is a qualified implementation of Horn clauses which has become important as a vehicle for Artificial Intelligence applications. In particular there is growing interest in its use for data base applications [3]. Since Prolog itself is not very convenient as a query language, various researchers have sought to develop other user interfaces to Prolog data bases. These include natural language interfaces [4] and Query-by-Example [5].

Query-by-Example (QBE) is a non-procedural data base query language developed by Zloof [6] in which queries are expressed by filling in skeleton tables with examples of the result required. In a human factors experiment conducted by Thomas and Gould [7] to determine the ease of use of data base query languages, the advantages of QBE over SQUARE and SEQUEL were clearly demonstrated. In particular they found that subjects using QBE required about one-third the training time, were somewhat faster in expressing queries and were about twice as accurate [7].

In view of this and the similarity between the syntax

of Prolog goals and QBE [8], an implementation of QBE interfacing with a logic data base has been realized in Prolog. Details of the implementation are given in [5].

Despite its simplicity and ease of use, QBE is relatively inflexible and makes no attempt at intelligent helpfulness or co-operativeness. This paper consider some ways in which the co-operativeness of QBE can be improved.

2. Incomplete queries

In QBE all queries must be expressed in full in a manner which reflects the way in which the data has been stored in the data base. However, the inexperienced or casual user may have difficulty in remembering the internal structure of the data and the way in which any particular query must be framed in order to reflect this. On the other hand the experienced user may find the process a little clumsy and look for short cuts. The idea of an incomplete query may appeal to either type of user.

In QBE any simple query which involves the join of two relations makes use of a common variable which occurs in one field of each of the two relations.

For example, given the data base in Appendix 1, suppose that the user wants to find the names of all suppliers who supply part number 2. The parts which he/she might enter are underlined "___". The entry might be:

suppliers	sno	sname	status	city
	S	p.N		

supplier_parts	sno	pno	qty
	S	2	

The common variable here which serves to join the two relations is S. Such a variable will be referred to as a link variable, and the fields of the two relations which are linked together (sno of suppliers and sno of supplier_parts) will be referred to as link fields.

In general there is no choice in the pair of link fields which can be used to join two relations together. For example, in the case of the relations suppliers and supplier_parts, the field sno of relation suppliers and field sno of relation supplier_parts are the only possible pair of fields which can be used to join these two relations.

In some cases it may not be possible to join relations directly and a join may only be effected via one or more intermediate relations.

For example, suppose that the user wishes to retrieve the names of all suppliers who supply at least one red part. The essential information in this query is:

suppliers	sno	sname	status	city
	p.N			

parts	pno	pname	colour	weight
			red	

although the complete query is:

suppliers	sno	sname	status	city
	S	p.N		

supplier_parts	sno	pno	qty
	S	X	

parts	pno	pname	colour	weight
	X		red	

where S and X are both link variables and supplier_parts is an intermediate relation.

Since in general link variables are not an essential part of a query but rather a result of the way in which data are stored in the system, it should be possible for the user to omit link variables from any query (together with any empty intermediate tables which may result). Any query in which one or more of the link variables have been omitted will be referred to as an incomplete query.

However, there is one snag with the omission of the link variables. Consider the query:

suppliers	sno	sname	status	city
		p.N		

supplier_parts	sno	pno	qty
		2	p.X

If this is treated as an incomplete query the system would attempt to link together these two requests. The result might be:

suppliers	sno	sname	status	city
	S	p.N		

supplier_parts	sno	pno	qty
	S	2	p.X

which would be interpreted as "print the name of each supplier who supplies part number 2 and the quantity supplied". On the other hand, the original query is sufficient in its own right being interpreted as "print the names of all suppliers and the quantities of part number 2 as supplied by different suppliers". The latter is a form of OR-query.

In general an incomplete query will have the same form as an OR-query and the system will be unable to distinguish

between the two. Thus it must be assumed that an incomplete query will not involve an OR-condition and that the user will indicate when an incomplete query has been issued.

In the next section the underlying data structures and the general approach to implementation of incomplete queries are discussed.

3. Implementation of incomplete queries

If a user wishes to issue an incomplete query, the query is entered in exactly the same way as any other query except that a different key (for example, a special function key in the keyboard) is used to signal the end of the query.

When the system is presented with an incomplete query, it attempts to link together the separate parts of the request. If it succeeds in finding appropriate links, the resulting query will be displayed in full to the user. If this resulting query satisfies the user, he/she indicates acceptance of the query by pressing the key normally used at the end of a complete query; if it is not what the user wants, a different key is used to indicate to the system to continue its search. If no suitable links can be found, the system reports this to the user.

To illustrate this, consider a request for the names of any suppliers who supply widgets and to whom one does not owe money at the present moment. IQ and CQ are used to denote the keys corresponding to Incomplete Query and

Complete Query respectively. The dialogue might be as follows (commentary is in /* ... */ brackets):

suppliers	sno	sname	status	city
	p.N			

parts	pno	pname	colour	weight
	widget			

supplier_balance	sno	amountowed
	X::X=<0	

```
IQ  /* signals the end of an incomplete query */
—
```

The infix operator "::", is used for syntactic convenience only and is to be read as "such that".

In response to this the system might display:

suppliers	sno	sname	status	city
	A	p.N		

supplier_parts	sno	pno	qty
	A	B	

parts	pno	pname	colour	weight
	B	widget		

supplier_balance	sno	amountowed
	A	X::X=<0

3.1. Formal specification of links

The data structure used to represent the data base relations and the connections between the relations is an undirected graph.

Fig 1 is a diagrammatic representation of the graph representing the links of the data base in Appendix 1. Every data base relation is represented by a vertex or node, called a relation node, and for every two nodes, if the same attribute occurs in both relations, an edge will connect the pair. This edge is labelled with the pair of attribute names from the two relations.

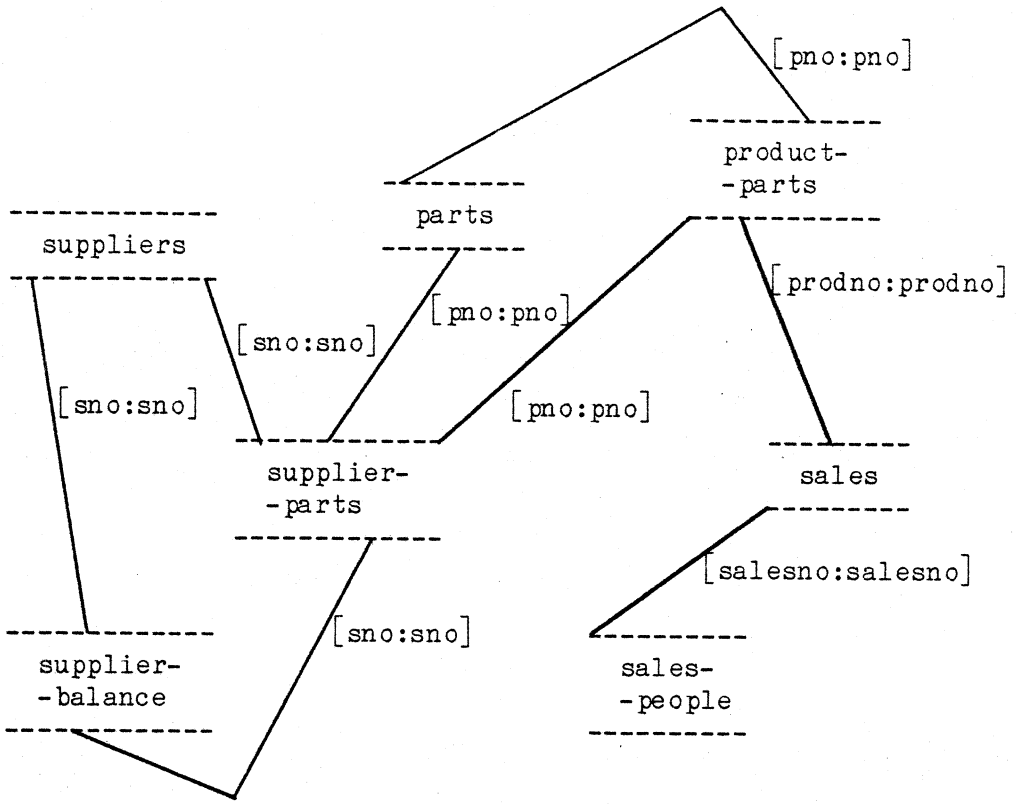


Fig 1 - The graph structure representing the link dictionary for the data base given in Appendix 1.

This information is represented within the query language system by a set of clauses of the form:

```
link(RELATION NAME 1, RELATION NAME 2,
      [
         ORDERED SEQUENCE OF LINK FIELDS OF RELATION 1:
         ORDERED SEQUENCE OF LINK FIELDS OF RELATION 2
      ]).

setofnodes(GRAPH NAME,
           [
              SET OF GRAPH NODES
           ]).
```

In Appendix 2 the link dictionary for the data base in Appendix 1 is given. Also presented is the Prolog program for searching for a path linking any pair of relations in

the data base.

The link dictionary described can be accessed by the user through the normal query mechanism, thus enabling the user to examine or update the structure of the data in the data base.

For example, suppose that the user wants to find which relations are linked with which . The entry might be:

```
p.links   |
          |
----------|-----------

CQ  /* signals the end of a complete query */
--
```

The system will respond by displaying for each relation R a list of relations linked to R, e.g.

```
links    | supplier_parts
---------|----------------
         | supplier_balance
         | suppliers
         | parts
         | product_parts

        ...
        ...
        ...
```

3.2. Handling join conditions

In order to handle join conditions the system determines the number N of unlinked components of the request and then seeks to establish the paths linking them together.

For example, suppose that the user wants to find the names of any suppliers to whom no money is owed at the present moment and who supply part number 1023. The entry might be:

```
suppliers    | sno   sname   status   city
-------------|---------------------------
             | X     p.N
               —     ——

supplier_balance  | sno   amountowed
------------------|-----------------
                  | X     A::A=<0
                    —     ———————

product_parts  | prodno   pno   noreqd
---------------|----------------------
               | 1023
                 ————

IQ  /* signals the end of an incomplete query */
—
```

where the user has partially specified the links by using the variable X to link relation suppliers with relation supplier_balance.

Given a query which contains join conditions, the paths linking the different components of the whole request may be established using:

> (i) - relation merging; that is, if two relations x and y which form part of the query are related to each other through the join variables A1, A2, ..., An (n>=1), merge relations x and y by performing joins between relations x and

y. Repeat this operation until no further merges are possible.

(ii) - graph generation; that is, look for paths which connect the remaining unlinked components of the graph (these must involve intermediate relations).

Let join-relation be the relation obtained by the join of relation suppliers with relation supplier_balance. Then the graph for the unlinked components of the initial request is as shown in Fig 2.

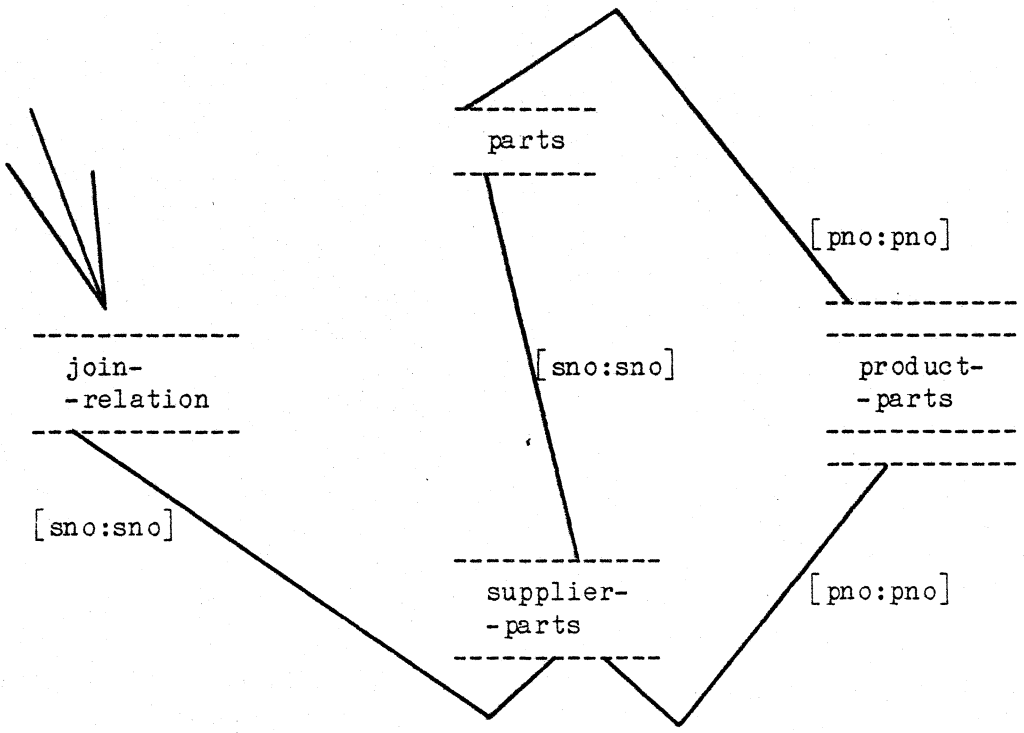


Fig 2 - Graph structure after merging.

The start node is indicated on the graph by an arrow, and double bars have been used to distinguish the final node. As a response, the system might display:

suppliers	sno	sname	status	city
	X	p.N		

supplier_balance	sno	amountowed
	X	A::A=<0

supplier_parts	sno	pno	qty
	X	Y	

product_parts	prodno	pno	noreqd
	1023	Y	

4. Incomplete updates

The ideas outlined in the previous section apply also to update operations.

For example if the user wishes to set the quantity to zero for all suppliers living in London, he/she might enter:

supplier_parts	sno	pno	qty
u			0

suppliers	sno	sname	status	city
				london

```
IQ  /* signals the end of an incomplete query */
```

to which the system will respond with:

supplier_parts	sno	pno	qty
u	Q		0

suppliers	sno	sname	status	city
	Q			london

In addition this link information may also be used in the case of update operations to ensure that when the user attempts to update a value in a link field of some relation, he/she is reminded of the possibility that the corresponding link field in some other relation may need to be updated too. In such a case the system might ask the user whether he/she wishes the same operation to be performed in the corresponding link field in the appropriate relation.

For example, if the user wishes to change the supplier number 13 to 3, the user might enter:

suppliers	sno	sname	status	city
u	3			
—	—			
	13			
	—			

```
CQ  /* signals the end of a complete query */
—
```

The system should then ask the user whether in addition he/she wishes to perform the following updates:

supplier_parts	sno	pno	qty
u	3		
	13		

supplier_balance	sno	amountowed
u	3	
	13	

In such case the user must indicate whether he wishes the additional update operations to be performed or rejected.

5. Queries which fail

In formulating a query a user inevitably makes certain presuppositions about the data present in the data base. These presuppositions are inherent in the information contained in the query and are an indication of what the user believes about the state of the information in the data base.

A data base query can be viewed either as requesting the selection of a subset (termed the response set) from a set of qualified instances in the data base, or as expressing some general belief about the data in the data base. In either case queries presented in QBE are translated into an intermediate meta language before being presented as a conjecture that a resolution-based theorem prover (e.g. Prolog) attempts to prove. This meta language is a graph structure, the nodes of which represent both data base relations and conditions imposed on the relation's

attribute(s).

The query graph is divided into connected subgraphs, each of which in itself constitutes a well-formed query in the meta language and is translated into a conjecture that can be presented to the theorem prover to be proved (i.e. each connected subgraph corresponds to a presupposition the user has made about the domain of discourse).

The next section discusses how the presuppositions inherent in these subgraphs can be used to provide a more co-operative response to users for both queries that request the selection of a subset of qualified instances in the data base and YES-NO queries.

5.1. Constructing corrective indirect answers

When dealing with queries requesting the selection of qualified instances in the data base (i.e. with queries defining a property of data base objects) consider the situation where the system fails to prove the conjecture (the initial query returns the empty set as an answer). In this case, on request from the user, the system will try to establish the user's presuppositions by translating each connected subgraph into a conjecture to be proved. This approach ensures that should a presupposition fail, an appropriate corrective indirect answer [9] will be returned to the user.

For example, suppose that the user wishes to retrieve

the numbers of all suppliers living in London who supply part number 2. The entry might be:

suppliers	sno	sname	status	city
	p.X			Y

supplier_parts	sno	pno	qty
	X	Z	

CONDITIONS
Z=2 and Y=london

```
CQ  /* signals the end of a complete query */
```

This query is based on the following presuppositions (i.e. the preconditions for the correctness of any direct answer):

(i) There are suppliers.
(ii) There are suppliers who supply parts.
(iii) There are suppliers supplying part number 2.
(iv) There are suppliers living in London.
(v) There are suppliers living in London who supply part number 2.

Should any of these presuppositions fail to be true, the system would, in general, respond with an empty list or "NULL". If, however, this query were addressed to a human being one might expect a more co-operative response which identifies the failing presupposition(s).

A complex query asking for the display of certain data items subject to a variety of retrieval conditions will be decomposed into a number of basic components in the meta language (i.e. connected subgraphs), each of which are acceptable queries in their own right. With each sub-query is associated a subset of the original set of presupposition(s). In the case of the above example, this can be represented diagramatically as:

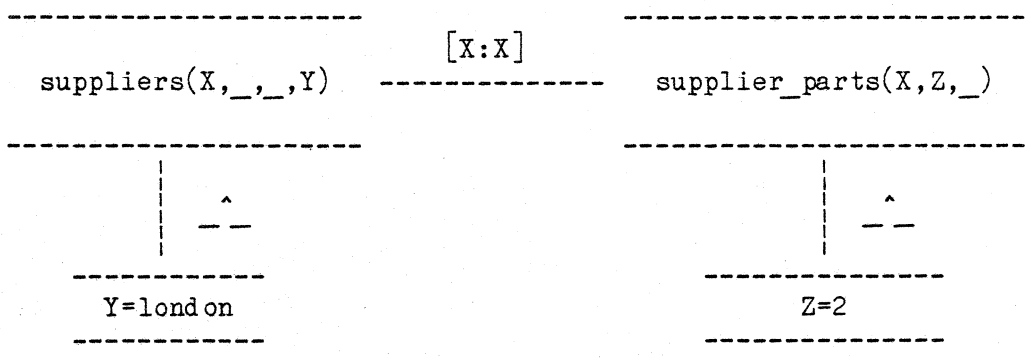


Fig 3 - The complete query.

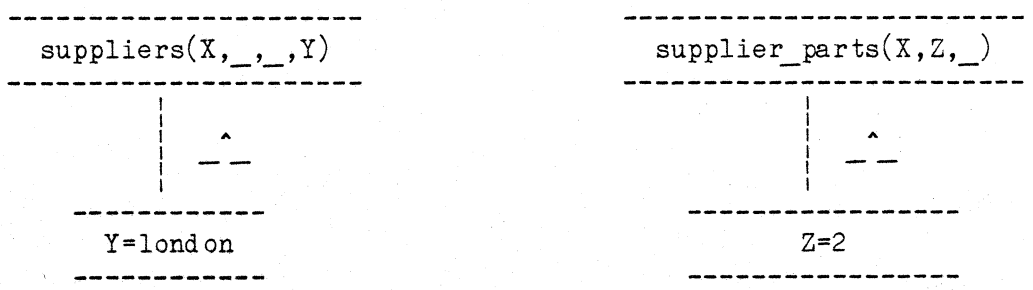


Fig 4 - Two first-level components

Fig 5 - Two second-level components

which will be translated by the system to yield:

```
<- suppliers(X, _, _, Y),
   Y=london,
   supplier_parts(X, Z, _),
   Z=2.
```

This clearly consists of two components:

```
<- suppliers(X, _, _, Y),
   Y=london.

<- supplier_parts(X, Z, _),
   Z=2.
```

each of these in turn depend on components:

```
<- suppliers(X, _, _, Y).

<- supplier_parts(X, Z, _).
```

In this case the system's response "NULL" will be produced only in the case where the top level query has failed but all sub-queries have succeeded. Otherwise the message "NULL-LOWER LEVEL QUERY FAILED" will be displayed.

On request the system will attempt to determine the cause of failure. If any sub-query fails and its failure contributes to the failure of the top level query, then:

- the failure of the sub-query will be reported back together with any other sub-query on the same level which

contributes to the failure of the top level query,

- any higher level failing sub-queries, not failing due to failure of component sub-queries, will be also reported back.

In the current implementation this is achieved by typing the keyword "WHY".

For example, if the query described above is presented to the system but the system fails to find any supplier living in London, it will respond with:

suppliers	sno	sname	status	city
	p.X			london

```
results: NULL  /* the empty set */
```

that is, the system recognizes the failure of the component:

```
<- suppliers(X, _, _, Y),
   Y=london.
```

and responds appropriately.

On the other hand, an OR-query fails if and only if all of its sub-queries fail. If one succeeds, the query as a whole succeeds, even if all the others fail. Such a situation might contribute to the misinterpretation by the user of the system's response due to the false assumptions made about the way the answer was inferred.

For example, suppose that the user wishes to retrieve the names of all suppliers who live in London or Paris. The entry might be:

suppliers	sno	sname	status	city
		p.X		london
		p.Y		paris

```
CQ  /* signals the end of a complete query */
--
```

to which the system's answer is:

results	sname
	jones
	blake

But it is known (see Appendix 1) that Jones and Blake both live in Paris, and that no one is in London. That is, the user can think of suppliers living in London and living in Paris to be correct, and carry on with a frustrating series of questions, or worse, misinterpret the system's response. To avoid such a situation, the user can, as soon as the answer has been displayed, request further information about the process used in the evaluating of the query. The result might be:

suppliers	sno	sname	status	city
		p.X		london

```
results: NULL  /* the empty set */
```

which indicates to the user that the presupposition that some suppliers were living in London is incorrect.

To the extent that update operations involve an initial request (query) aimed at locating certain qualified instances or tuples in the data base, a similar type of analysis as the one described above would apply in the case of failure.

In the case of a query which expresses some property of the data base as a whole, should the system fail to prove the conjecture, an attempt is made to prove the negation of the conjecture in order to answer "NO". Should the system fail to prove or disprove a given conjecture an answer of "DON'T KNOW" is returned to the user. This is the case when neither a "YES" nor "NO" answer is possible from the axioms in the data base.

If the answer is "NO" or "DON'T KNOW" an analysis of the presuppositions made might follow if requested.

6. Conclusions

Most query systems currently available respond to queries in a very literal manner, giving an answer to what the user actually asked for - no more and no less. Though the responses are literally correct, such rigidity can be very unhelpful at times, and a more flexible system is desirable. This flexibility in the interpretation of queries in a manner which is both natural and of benefit to the user is termed co-operativeness.

This paper outlines several ways in which the query

language QBE can be made more co-operative. These features have been added to a version of QBE implemented in Prolog, which is running under UNIX on both a PDP 11/34 and a DEC 10 system.

The main features of such a system are:

(i) A link dictionary has been implemented which contains information about the data base relations and the linkages between them. This facility was interfaced with the query facility to provide the user with the means to examine how the data in the data base are organized and how they should be accessed and used.

(ii) The system attempts to handle incomplete queries and updates by filling in link variables. This can be of use to casual users of the data base who do not have the details of the structure of the data base at their fingertips, as well as to experienced users who seek short cuts.

(iii) The system reminds users of possible side effects when updates are performed on link variables.

(iv) The system attempts to provide a helpful response when a complex query fails to give the user an indication of why it failed. The same tabular form is used to explain the reasoning it followed to arrive at the answer as that used to enter the initial request.

Acknowledgements

The work of one of the authors, J C NEVES, was supported by the Calouste Gulbenkian Foundation under grant 14/82 and by an ORS award from the Committee of Vice-Chancellors and Principals of the Universities of the United Kingdom.

J C NEVES is on leave from Minho University, Largo do Paco, 4700 BRAGA, PORTUGAL.

7. References

[1] T. Moto-Oka et al, Challenge for knowledge information processing systems, in: Fifth Generation Computer Systems (North-Holland, Amsterdam, 1982) 3-89.

[2] D. H. D. Warren, Implementing Prolog - Compiling Predicate Logic Programs. Technical Report 39 and 40, Department of Artificial Intelligence, University of Edinburgh, 1977.

[3] H. Gallaire and J. Minker (eds), Logic and data bases, Plenum Press, New York, 1978.

[4] F. C. N. Pereira and D. H. D. Warren, Definite clause grammars for language analysis - a survey of the formalism and a comparison with augmented transition networks, Artificial Intelligence 13 (1980) 231-278.

[5] J. C. Neves, S. O. Anderson and M. H. Williams, A Prolog implementation of Query-by-Example, in: Proceedings of the 7th International Computing Symposium, March 22-24, 1983, Nurnberg, Germany.

[6] M. M. Zloof, Query-by-Example: A data base language, IBM Systems Journal 16(4) (1977) 324-343.

[7] J. C. Thomas and J. D. Gould, A psychological study of Query-by-Example, Proc. National Computer Conference (1975) 439-445.

[8] M. H. van Emden, Computation and deductive information retrieval, in: Formal Description of Programming Concepts (North-Holland, Amsterdam, 1978).

[9] S. J. Kaplan, Co-operative responses from a portable natural language query system, Artificial Intelligence 19 (1982) 165-187.

Appendix 1: A simple business data base

The examples in this paper make use of the following relations:

(i) A relation called parts with attributes (columns): pno (part number), pname (part name), colour and weight.

(ii) A relation called suppliers with attributes: sno (supplier number), sname (supplier name), status and city.

(iii) A relation called supplier_parts with attributes: sno (supplier number), pno (part number) and qty (quantity supplied).

(iv) A relation called supplier_balance with attributes: sno (supplier number) and amountowed.

(v) A relation called sales_people with attributes: salesno (sales number) and salesname.

(vi) A relation called product_parts with attributes: prodno (product number), pno (part number) and noreqd (number of parts required).

(vii) A relation called sales with attributes: salesno (sales number), prodno (product number) and qtysold (quantity sold).

Typical values of these relations are as follows:

parts	pno	pname	colour	weight
	1	nut	red	12
	2	bolt	green	17
	3	screw	blue	17
	4	screw	red	14
	5	cam	blue	12
	6	cog	red	19

Table 1.1 - The parts relation

suppliers	sno	sname	status	city
	1	smith	20	vienna
	2	jones	10	paris
	3	blake	30	paris
	4	clark	20	vienna
	5	adams	15	athens

Table 1.2 - The suppliers relation

supplier_parts	sno	pno	qty
	1	1	300
	1	2	200
	1	3	400
	1	4	200
	1	5	100
	1	6	100
	2	1	300
	2	2	400
	3	2	200
	4	2	200
	4	4	300
	4	5	400

Table 1.3 - The supplier_parts relation

supplier_balance	sno	amountowed
	1	100
	2	90
	3	0
	4	0
	5	145

Table 1.4 - The supplier_balance relation

sales_people	salesno	salesname
	1	flanagan
	2	ellis
	3	smith
	4	schafer

Table 1.5 - The sales_people relation

product_parts	prodno	pno	noreqd
	1027	1	350
	1023	1	200
	1028	1	100
	1033	3	275
	1040	4	435
	1072	5	555
	1045	2	315
	2001	6	125
	1067	5	111

Table 1.6 - The product_parts relation

sales	salesno	prodno	qtysold
	1	1023	100
	1	1027	45
	2	1028	40
	3	1033	150
	3	1040	75
	4	1072	20

Table 1.7 - The sales relation

Appendix 2: The link dictionary for the data base in Appendix 1.

The link dictionary for the data base in Appendix 1.

```
link(supplier_parts, supplier_balance, [sno]:[sno]).
link(suppliers, supplier_balance, [sno]:[sno]).
link(supplier_parts, product_parts, [pno]:[pno]).
link(parts, product_parts, [pno]:[pno]).
link(sales, product_parts, [prodno]:[prodno]).
link(sales_people, sales, [salesno]:[salesno]).
link(supplier_parts, suppliers, [sno]:[sno]).
link(supplier_parts, parts, [pno]:[pno]).

setofnodes(graph, [supplier_balance, product_parts, sales,
                   sales_people, suppliers, parts,
                   supplier_parts]).
```

The Prolog program for searching for a path linking any pair of relations in the data base:

```
                ?- op(40, xfx, :).

                /* declare ":" infix operator   */


clause 1        path(GRAPH, X, Y, PATH) <-
                setofnodes(GRAPH, SET),
                member(X, SET),
                member(Y, SET),
                walk(GRAPH, [X], Y, PATH).

clause 2    walk(GRAPH, [Y|L], Y, [Y|L]).
clause 3    walk(GRAPH, [X|L], Y, PATH) <-
                (
                   link(X, Z, _);
                      link(Z, X, _)
                ),
                not(member(Z,L)),
                walk(GRAPH, [Z,X|L], Y, PATH).

clause 4    member(X, [X|_]).
clause 5    member(X, [_|Y]) <-
                member(X, Y).
```

PROGRAPH AS AN ENVIRONMENT FOR PROLOG DATA BASE APPLICATIONS

by T. Pietrzykowski
Acadia University
Wolfville, Nova Scotia
Canada BOP IXO

ABSTRACT

This Paper presents a specialized data base model of a newly developed functional programming language with graphical front PROGRAPH, and discusses the advantages of using it as a host from PROLOG. On the basis of an example there is a description of a method of converting (via PROLOG) an arbitrary query into a wff containing only data base and computational predicates. Afterwards such a wff is formulated in PROGRAPH and computed (again a series of examples and a method provided).

1. Introduction

Logic programming approach proves, beyond any doubt, to be the most universal and flexible mechanism for data base queries ([6], [7] etc.). Its power becomes particularly visible when a query involves a more advanced conceptual structure and requires non trival computations. However, for the full success of logic programing in this area is handicapped by the following shortcomings:

(i) awkwardness of executing queries, which involve universal quantifcation. The usual method of converting them into negated existential qualification of negated formula ([5]) leads often to considerable inefficiency of the search procedure. Moreover, universal

qualification generally accompanies implication inside of the query what in term produces non Horn clauses.

(ii) communication with data base only via unification of the unit clauses brings various dilemnas: the unit clauses with a small number of variables increase flexibility in formulating queries but also increase depth of deduction while unit clauses with a large number of variables may dangerously expend the size of the data base representation and conseqently leads to the lose of efficiency.

(iii) relational data base model (which becomes a consequence of the unit clause approach discussed in (ii)) may causes lose of important information about the overall structure of data base which could be used as valuable heuristical guidelines for an efficient search procedure which is the foundation of the computation of queries.

In the following we shall propose an alternative approach which deals with the data base model as well as with the computation of queries. It will be based on a newly developed programming language PROGRAPH ([3], [4]) which will be used as a host system for logic programing, particularly in the area of interface with a data base.

2. PROGRAPH Data Base.

We will start our presentation from the description of the data base model. It can be viewed as a combination of a particular case of the relational model with some network model influences. However, these similarities may be more misleading then helpful and the best way would be to consider it on its own.

We shall present three different but strictly equivalent definitions of the model. Later we shall refer to any of them: whichever appear to be most convenient.

A. Directed graph model.

The PROGRAPH data base can be defined as a directed graph with labelled arcs. We shall call it graph database or shortly GD. We assume that the graph is connected. The set of nodes N of GD is partitioned into two distinct categories: abstract records and data

records. While abstract records are abstract elements without any particular qualities the data records are trees where leaves are elementary data of basic types like integer, real, boolean, strings of characters, etc.

Among the abstract records there is one specifically distinguished called root and denoted as  . The arcs GD are labelled by a string of characters which are called attributes.

B. Functional Model.

This model is called functional database of FD. It consists of a set elements identical to nodes N of GD and a set of partial multivalued functions with domains and ranges in N. Each of these functions corresponds to a distinct attribute of GD in a one-to-one manner: if F is a function of FD then the domain of F is a subset of N consisting of all nodes where any arc of GD with the attribute F originates while range is the set of all nodes where such an arc ends. The connectivity condition can be easily expressed in terms of FD.

C. Relational Model.

This model called relational database (RD) is a simple variation of the functional model. It is defined as a set of partial binary relations over N. If P is a relation of RD then P(x,y) holds iff y P(x) where P(x) is defined in terms of FD. (The notational ambiguity is hopefully resolved by the reader).

Now we shall present an example of a PROGRAPH database using both GD and RD models. Construction of the FD model for this example is left for the reader

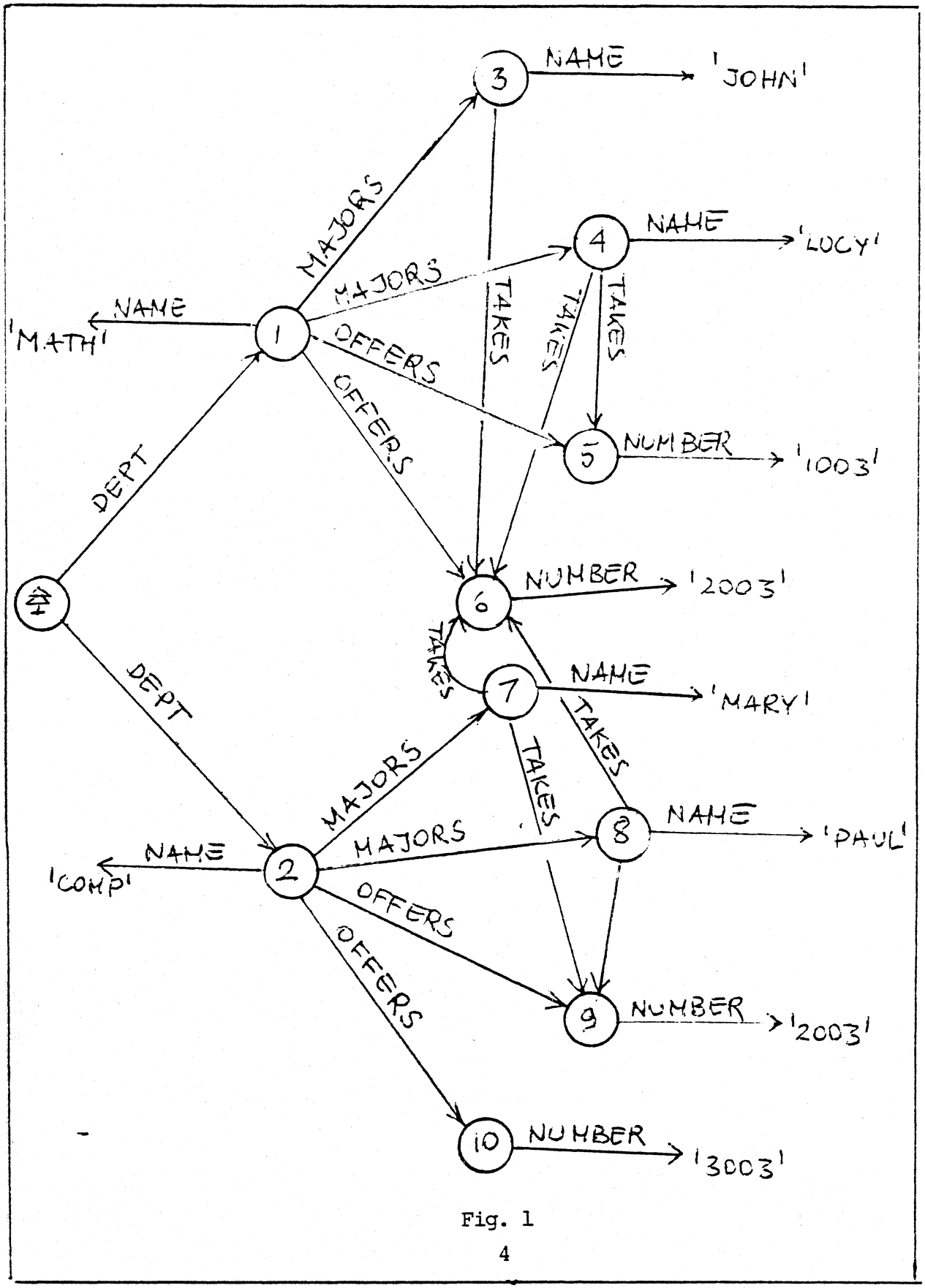

3
NAME
'JOHN'
MAJORS
4
NAME
'LUCY'
MAJORS
NAME
'MATH'
1
TAKES
TAKES
TAKES
OFFERS
OFFERS
5
NUMBER
'1003'
DEPT
OFFERS
NUMBER
'2003'
6
TAKES
7
NAME
'MARY'
DEPT
TAKES
TAKES
MAJORS
MAJORS
8
NAME
'PAUL'
NAME
'COMP'
2
OFFERS
OFFERS
9
NUMBER
'2003'
10
NUMBER
'3003'

Fig. 1

This GD represents a structure composed of departments, students majoring, courses offered, courses taken as well as the names of departments and students and numbers of courses. The integers inside of nodes have no semantic significance and are used only as a way of referring to individual abstract records nodes in further discussion. For example: nodes 1, 2 correspond to departments: 1 to Mathematics while 2 to Computer Science 3, 4, 7, 8 are students with names JOHN, LUCY, MARY and PAUL respectively. The data records have no identifying numbers and we refer to them via the corresponding data.

A useful way of looking at the above GD is to compress it into a map as presented below.

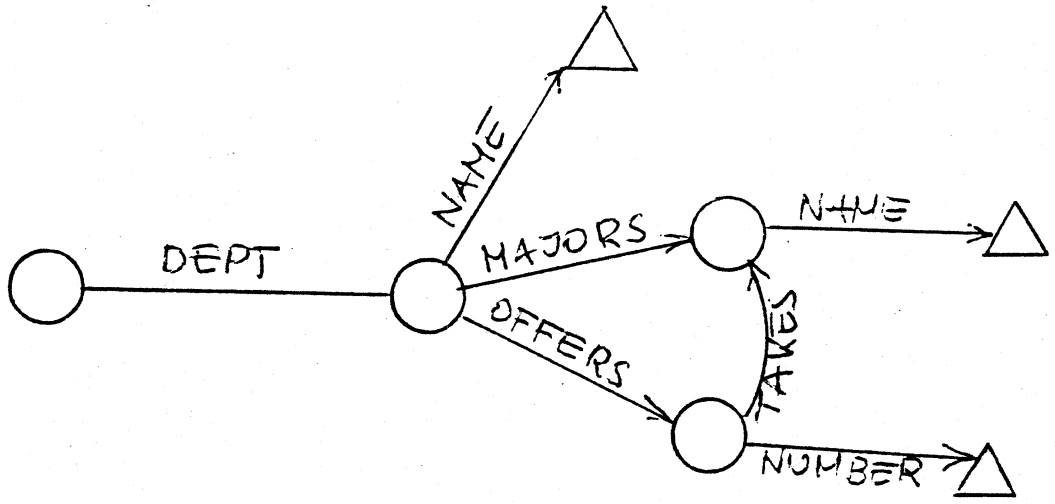


Fig. 2

The map is obtained by recursively collapsing all the arcs with the same attribute originating in a node into one, following by collapsing the corresponding end nodes of these arcs. The circle nodes of the map corresponds to sets of abstract records of GD while triangle nodes correspond to sets of data records.

Now we represent database of Fig. 1 as RD in the form of binary tables. We use the numbers attached to appropriate nodes as entries to the tables above while the heading of a table corresponds to a relations name (attribute).

DEPT	
	1
	2

MAJORS	
1	3
1	4
2	7
2	8

OFFERS	
1	5
1	6
2	9
2	10

TAKES	
3	6
4	5
4	6
7	6
7	9
8	6
8	9

NAME	
1	MATH
2	COMP
3	JOHN
4	LUCY
7	MARY
8	PAUL

NUMBER	
5	1003
6	2003
9	2003
10	3003

Fig. 3

It is easy to note that the graphical structure presented on Fig. 1 and even more clearly on Fig. 2 contains valuable information on efficient implementing the data base. It suggests keys, access structures and storage allocation. In contrast in the flat tables of the relational model described on Fig. 3 this potentially useful information is lost: it can only be recovered by converting the tables into a graphical or functional structure.

3. Combining logic programming with PROGRAPH.

In the following we shall present a proposal on how to deal with problems of logic programming mentioned in the Introduction. Our aproach will use the PROGRAPH data base model and programming technique. Our presentation will be based on examples.

Let us consider the following query:
"are all the students regular, where regular means faithful but not overzealous? A faithful student takes at least one course from the department in which he/she majors, while overzealous takes all the

courses from such a department".

Now we shall present the same query as a mixture of predicate logic and PROLOG:

(i) $\forall x$ **regular** (x)

(ii) **regular**(x) := **faithful**(x), **notoverzealous**(x)

(iii) **faithful**(x) := $\exists y \exists z[(DEPT(\text{令},y) \wedge MAJORS(y,x)) \supset (TAKES(x,z) \wedge OFFERS(y,z))]$

In the above query and following it definitions, let us distinguish two types of predicates: defined predicates like **regular**, **faithful** and **notoverzealous** and evaluation predicates, like DEPT, MAJORS, TAKES and OFFERS. The former we denote by using bold face letters while the latter by capitals. The defined predicates occur, at least once, on the left hand side of PROLOG expressions while evaluation predicates are attributes of the PROGRAPH data base or computation predicates like $x \geq y$ or $x+y=z$ (absent in our example).

Now we are ready to describe the processing of the query. We apply PROLOG mechanism to replace all occurences of the defined predicates, starting with the actual query in formula (i). (Let us note that the query is not negated or skolemized.)

These replacements will follow the rules of logic programming with the understanding that occurrences of universally qualified variables are treated as constants. Substituting (ii) into (i) with the appropriate unification, we obtain:

(v) $\forall x$(**faithful**(x) $\wedge$ **notoverzealous**(x)).

At that moment the comma ',' separating the two subgoals is converted into conjunction (' $\wedge$'). Now we continue our activity substituting into (v) the formulae (iii) and (iv) to obtain, after easy optimization:

(vi) $\forall x \exists y[(DEPT(\text{令},y) \wedge MAJORS(y,x)) \supset (\exists z(TAKES(x,z) \wedge OFFERS(y,z)) \wedge \exists u(TAKES(x,u) \wedge \neg OFFERS(y,u))]$

Let us note that (vi) does not contain any defined predicates and PROLOG phase of processing is therefore terminated.

In general the situation is more involved because in the presence of recursive definitions such a state cannot be achieved, but it is not a new phenomenon: PROLOG will deal with it in the same way as it usually does with recursion. It should be mentioned that the whole mechanism described above can be without the difficulties implemented in PROLOG.

Before we move to the next stage of producing a prograph corresponding to the formula (vi) let us provide some information about PROGRAPH.

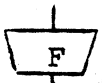
PROGRAPH is a programming functional language with a graphical front. It follows the direction of the Graphical Programming Language, GPL [2] developed at the University of Utah and dedicated to their data flow computer DDM 1 [1]. However, PROGRAPH goes much further then GPL allowing: compose operation, introduction of user defined subroutine, explicit indication of possible parallelism of computation and what is most important, it provides a mechanism for database access and update activities, which does not violate the functional character of the language. An experimental version of PROGRAPH is currently implemented on PERQ graphics station.

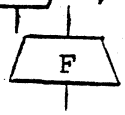
A PROGRAPH equivalent of 'program' is called 'prograph'. Generally speaking a prograph is a network of boxes connected by wires. A box, corresponds to a specific operation provided by the system (called primitive) or defined by a user. Such an operation is performed on datas supplied to its input and the results are delivered as outputs. The wires naturally connect inputs and outputs of distinct boxes without producing loops.

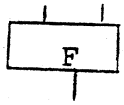
Now we shall introduce a few PROGRAPH primitives, necessary to present our query as a prograph.

Let F be an attribute of a PROGRAPH data base which we shall interpret, for the moment, as a binary relation. Let X be sets of nodes of data base applied respectively to input - top wire and Y resulting from output - bottom wire. As a matter of fact, inputs always are provided by top wires while outputs always are delivered by the bottom

ones.

The box 

, called access, means that Y= F[X] (in functional notation) while 

, inverse access, means $Y = F^{-1}[X]$. Now let x, y be single records provided as inputs of the box 

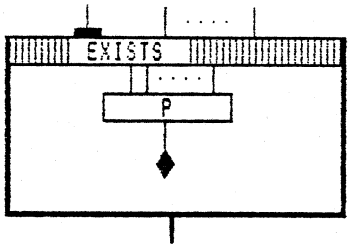
called application. Then the output z = F(x,y) (using relational notation). In this case the output is obviously of the type boolean, however PROGRAPH does not require specification of types of datas.

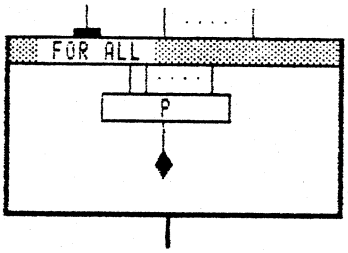
Let us introduce two obvious primitives: 

and 

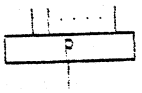
The oval boxes are used here only for visual effect so the user can easily distinguish logical operations.

Finally, we shall present two so-called composed operations (all the above ones are simple): EXISTS and FOR ALL.





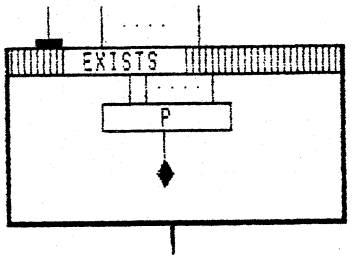
The number of inputs can vary while there must be one and only one called multiple which is signified by the  . It should be mentioned that multiple input does not have to be first to the left but it must not be more than one such input. It should be mentioned that the multiple input does not have to be first to the left.



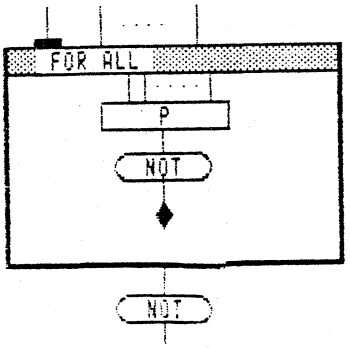
denotes an arbitrary prograph with k+1 inputs ($k \geq 0$) and one output of the type boolean. The semantics of these operations is: if $x, y, ,\ldots, y_k$ are values of inputs then the values of outputs are respectively:

$$\exists\, x(\, x \in X \wedge P(x, y, ,\ldots, y_k) \text{ and } \forall\, x(x \in X \supset P(x, y, ,\ldots y_k))$$

It is worth to note that the PROGRAPH definitions of quantifiers satisfy the basic properties of predicate logic: that is



is equivalent to



Now we are ready to present the PROGRAPH equivalent of the formula (vi)

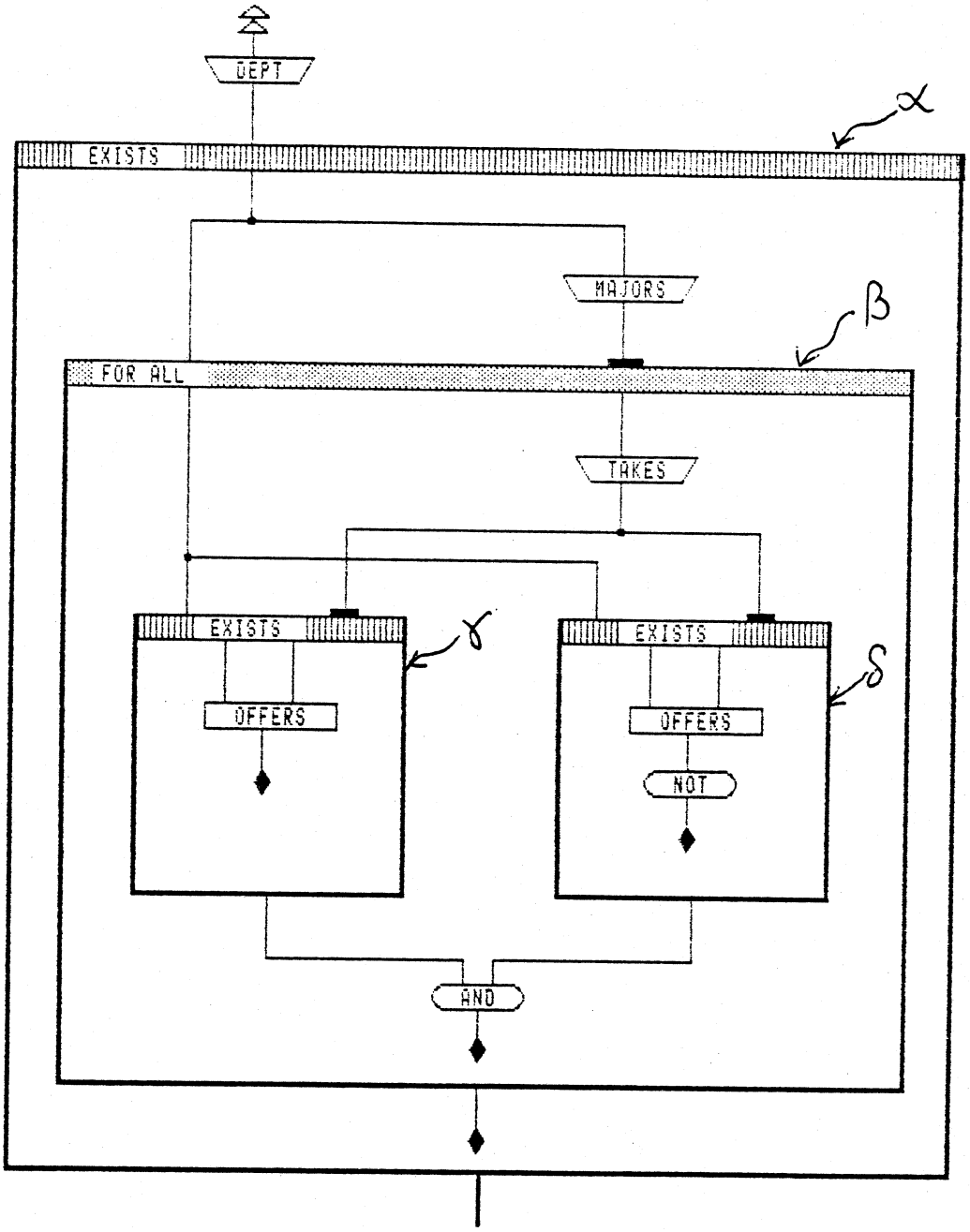


Fig. 4

Note that letters α, β, γ and δ are not part of the PROGRAPH description: they are introduced as references to appropriate EXISTS and FOR ALL boxes.

Now we shall present an informal description of how the prograph of Fig. 4 is derived.

First let us construct the following graphics presentation of the formula (VI) called outline which will be useful for our explanation.

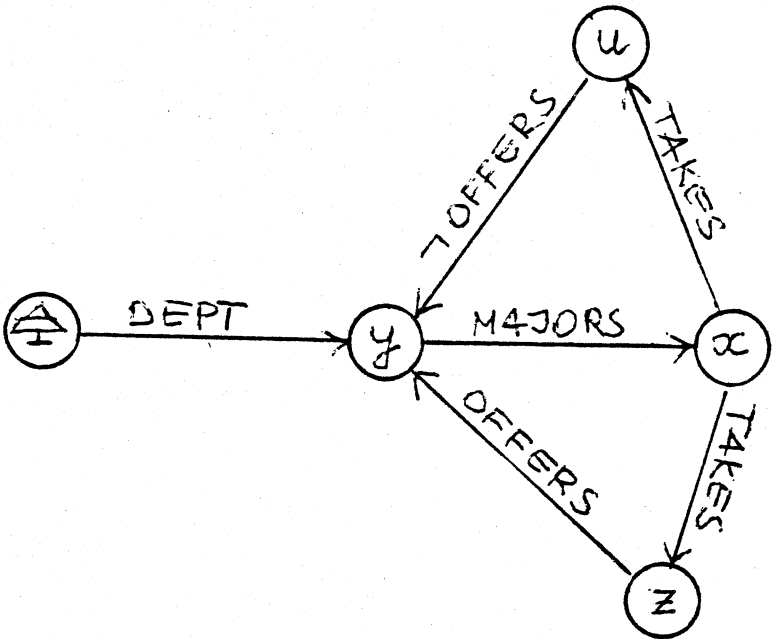


Fig. 5

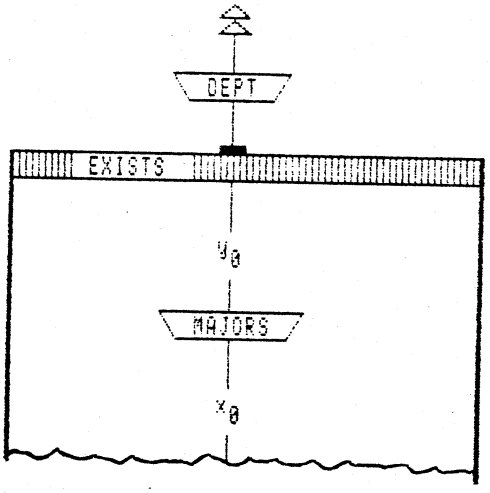
The outline is a directed graph with nodes corresponding to distinct variables in formula (VI). The labelled arcs correspond to predicates (or negated predicates) of (VI) in such a way that R labels arc originating in the node m and ending in n iff R(m,n) occurs in (VI). In order to derive the prograph of Fig. 4 we introduced a partial order among the arcs the outline which is imposed in natural way by the directions of arcs.

Now we can proceeded with constructing the prograph starting with a minimal arc (in this case: DEPT) and create:

Then we progress along outline and arrive at the end node of this arc (in this case node y), and create EXISTS box α since y is existentially quantified in (VI).

We proceed in an analogous manner with arc MAJORS: we introduce and create β box FOR ALL.

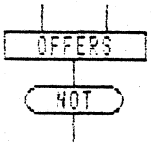
The rational behind the box is somewhat more complex so we will provide the reader with some additional explanation. Let us consider the top of the prograph of Fig. 5:



where y_o is the input to MAJOR and x_o the output. Obviously, y_o satisfies DEPT(, y_o) and $x \in x_o$ iff MAJORS(y_o,x). Therefore $x \in Xo$ iff DEPT(, y_o) $\wedge$ MAJORS(y_o,x) so in view of formula (vi) and definition of semantics of operation FOR ALL the introduction of the box is justified. Following the ordering of outline on Fig. 5 we arrive in node x and note that there are 2 arcs originating in x, both labelled TAKES. Therefore we introduce TAKES and branch the output. The corresponding branches are directed to EXISTS boxes and respectively, which corresponds to z and u, variables of Fig. 5. Now both arcs OFFERS and $\neg$ OFFERS end in the node y (already traversed). In this case we fill the boxes γ and δ with the

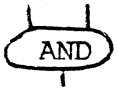
operations:

and



respectively

It is worth noticing that the first input wire corresponding to y variable originates in the box α, has been transmitted into β (first input wire) and branches there to arrive as first input to γ and δ respectively. Finally outputs of boxes γ and δ are joined as inputs to



box according to the conjunction of both existential subexpressions $\exists z(...)$ and $\exists u(...)$ in formula (VI).

The above description of an algorithm for producing a prograph from a well formed formula, as we mentioned already, is fairly infomal and sketchy. However, there is a formal algorithm performing this task which is unfortunately too lengthy to be described here and will be a subject of a separate publication.

To further convince the reader that the proposed approach is useful, we will present two more examples of data base queries and their PROGRAPH representation.

First query:

'does exist a course offered by the department of mathematics such that every student majoring in math takes this course'

Here is this query presented as a iff formula of predicate logic:

(viii) $\exists x \exists z \forall y[(DEPT(\text{⛪},x) \wedge NAME(x,MATH) \wedge MAJORS(x,y)) \supset (OFFERS(x,Z)\ TAKES(y,z))]$

Given below is an equivalent PROGRAPH formulation:

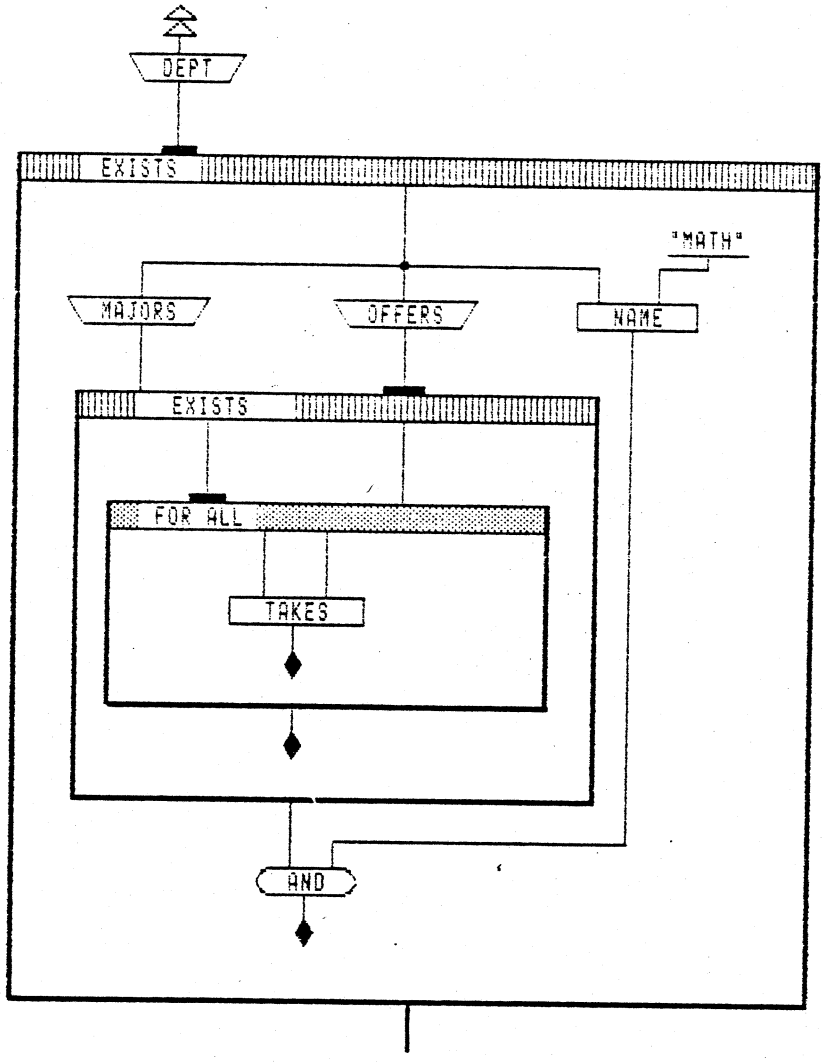


Fig. 6

Now we shall present a PROGRAPH version of the formula (viii) with a second and third quantifier reversed, so the prefix looks as follows: $\exists x \forall y \exists z$ and the matrix is unchanged.

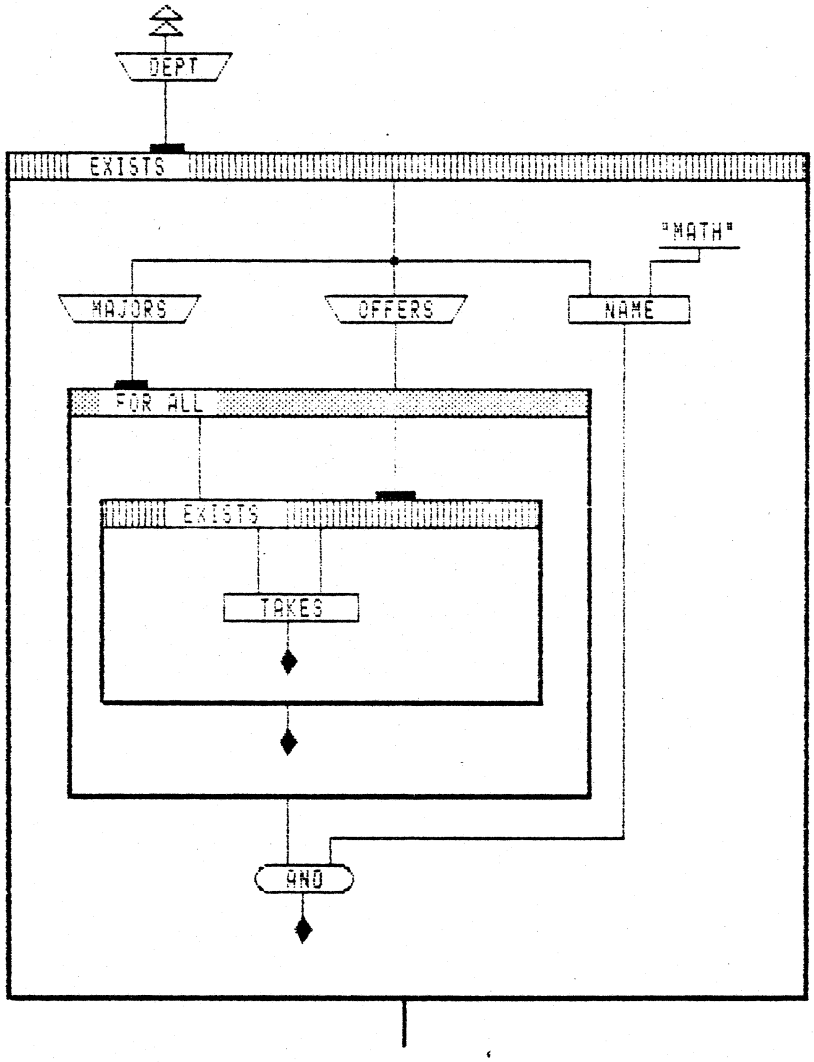


Fig. 7

The reader is encouraged, to find how the reversal of qualifiers effects the changes in corresponding prographs.

4. Conluding Remarks.

The presented results have a preliminary character, but in the author's opinion, leave no doubt that the approach is worth persuing. Since an experimental version of PROGRAPH is already functioning on a PERQ graphics station we intend to use it for a thorough series of experiments with a variety of data base queries formulated in terms of

PROGRAPH. The next stage will be to introduce an interface with a PROLOG implementation (unfortunately such one is not, at present, available on PERQ). This can be achieved by establishing appropriate communication with another computer or porting PROGRAPH onto a computer system where PROLOG is available. Finally, we would like to experiment with the combination of both as a uniform environment.

BIBLIOGRAPHY

[1] Davis, A. L., Keller, R. M., "Data Flow Program Graphs", IEEE Computer, Feb 1982, pp. 26-41.

[2] GPL Programming Manual, CS Dept., University of Utah, July, 1981.

[3] Pietrzykowski, T., "Programming Language PROGRAPH: Yet Another Application of Graphics" (with S. Matwin and T. Muldner), Graphics Interface 83 Conference, Edmonton, May 1983.

[4] Pietrzykowski, T., "Report on a Functional Language with a Graphical Front PROGRAPH: (with S. Matwin and T. Muldner), Research Notes, CS 83 02, School of Computer Science, Acadia University, 1983.

[5] Sato, T., "Negation and Semantics of PROLOG programs", Proceedings of 1st International Workshop on Logic Programming, Marseille, 1982.

[6] Van Emden, M. H., "Computation and Deductive Information Retrieval" in "Formal Description of Programming Concepts", North Holland, 1978.

[7] Warren, D., "Efficient Processing of Interactive Relational Data Base Queries Expressed in Logic", Proceedings of Conference on Very Large Data Basis, 1981.

Relational Data Bases

"à la carte"

Miguel Filgueiras

Luis Moniz Pereira

Nucleo de Inteligencia Artificial
Departamento de Informatica
Universidade Nova de Lisboa
Quinta da Torre
2825 Monte da Caparica
PORTUGAL

Abstract

We have developed a general purpose program (written in Prolog) which uses information gathered interactively from the user to generate specific menu based consultation programs, tailored to suit the relational data base and access requirements of each application. Every menu allows for quite general relational queries, comprising universal and existential quantifications, conjunctions, disjunctions, and negation as non-provability. Some of the relational data base access concepts employed concern implicit fields, special access predicates, references to text strings stored on disk, finding complete descriptions from partial ones, etc. .

We claim the great usefulness of this program : for those who have data to store and retrieve the only work is to plan a relational data base; the consultation program is almost instantly made.

The use of Prolog was paramount for the ease of design and implementation of this system.

Introduction

We have developed a general purpose program (written in Prolog) which uses information gathered interactively from the user to generate specific menu based consultation programs, tailored to suit the relational data base and access requirements of each application. Every menu allows for quite general relational queries, comprising universal and existential quantifications, conjunctions, disjunctions, and negation as non-provability. Some of the relational data base access concepts employed concern implicit fields, special access predicates, references to text strings stored on disk, finding complete descriptions from partial ones, etc. .

We claim the great usefulness of this program : for those who have data to store and retrieve the only work is to plan a relational data base; the consultation program is almost instantly made.

The use of Prolog was paramount for the ease of design and implementation of this system. Indeed, Prolog as language [W. Clocksin, C. Mellish 81] comprises in itself relational queries, and relational data bases are inherent to it. Additionally, it incorporates a search strategy for data retrieval, besides being a powerful symbol manipulating language on its own. Thus it is ideally suited for piecewise program generation ; this is so because Prolog clauses are extremely modular and need not have any side-effects. Consequently, the application dependent clauses of the consultation program are simply added to its application independent core.

Furthermore, queries are easily built along successive menu steps because Prolog clauses do not have to return complete data structures, but may cooperate instead to their successive refinement.

The user relational data base is just a Prolog subprogram, in the form of unit clauses, which is added to the consultation part. Updating is simply performed with an editor, which in some systems may be called from within Prolog. The criticism that the address space puts a limit on the data base size is waved in the 32-bit address machines. In smaller address space machines, one can set up several Prolog Jobs to hold data base parts, and have them communicate through a message queue handler. We have done so on a PDP 11/23 running under RT11-XM.

Basic Notions

Data are supplied, as in all relational data bases, as n-tuples of arguments of relations (or predicates), and stored as Prolog unit clauses. Each argument (or field) has a meaning dependent on the predicate and argument position.

For print out purposes, a heading for each field is requested from the user. The consultation program assumes identical headings correspond to similar fields, for all data base relations.

A distinction is drawn between '**output-only**' and '**both-way**' arguments : the former are only used for output and cannot be instantiated in queries, the latter can be used for data retrieval as well, and so can be instantiated in queries.

We have developed some optional features to increase data base compactness and improve access. They are :

- **References to texts.** Sometimes there are non-formatted informations that are best kept as texts. We use fields of the form **t(File, Number)** to refer to a text under the given **Number** in the given **File**. Such fields are viewed as **output-only** and we have a special predicate (in Prolog) to retrieve such texts.

- **Implicit fields.** It is useful for derived information (implicit fields) to be built from actual fields of a data base predicate, so as to avoid duplication. To this end we provide two different implementations. They should be chosen according to how often the derived information is required. For the implicit fields frequently used, an **interface predicate** is created that calls the corresponding data base one and builds all such fields. For infrequently used implicit fields, **ancillary conditions** are employed to define them. Each such condition is only activated whenever the correspondent implicit field is required. A data base predicate may be augmented with both kinds of implicit fields.

- **Special access predicates.** Information may need some preprocessing before being output or retrieved from an explicit or implicit field. We tackle this by allowing special user defined predicates to be called when accessing such fields. As an example consider the case of lists. One may be interested in obtaining not a list but one of its elements; pretty-printing may also be required.

To build a query one needs to know the type of operation envisaged, the logical connectives and predicates involved, as well as a specification of which fields have an input value imposed and which are to be output. Additionally, a request can be made for similar fields to hold the same value, or for a set of answers to be partitioned relative to the different values of one or more fields.

The first menu presented to the user offers a choice of quantifier/operations as well as calls to subprograms independent from the consultation program proper (e.g. the one that finds complete designations from partial ones - the '**oracle**'). Examples of quantifiers are

one **all** **how many**

and of operations on numeric fields are

sum **mean** **least value** **greatest value**

Next, for describing the information in question, the user is presented, in a first stage, with two more menus. One allows the choice of a data base predicate (or interface predicate if any), and of a combination of its fields and their mode of access. Another menu follows, to select between launching the query to obtain an answer (the specification is assumed complete) or to connect the partial specification to what follows with an **and / andnot / or / ornot** operator. The completion of the specification is then resumed from the first stage.

The specification of any field may be a combination of four modes :

- a value is input
- a value is to be output
- output is to be grouped according to its different values
- the information in this field must match the information in similar field(s) (i.e., those with identical output headings) occuring in predicates already incorporated in the query

All non-contradictory combinations of these can be made, the consultation program rejecting any inconsistency : of course one cannot give two different values to be matched

simultaneously, but can give a value and ask it to be output as well. **Output-only** fields, too, cannot be input a value.

The following section thoroughly examplifies these features.

A Consultation Program

We now describe in some detail a real data base system made under a contract with **JNICT** - 'Junta Nacional de Investigacao Cientifica e Tecnologica' (National Science and Technology Research Council) - regarding data on **FACC** - 'Fundo de Apoio 'a Comunidade Cientifica' (Scientific Community Support Fund) - concerning research centers (about **200**), their organics (one per center and per year), and applications for funding (about **500** per year) [L. Moniz Pereira, M. Filgueiras 82].

The extensional data base predicates were designed as follows :

center(Number_c, Initials, Sector, District, Info_c)

organic(Number_c, Year, Director_Title, Director_Name, Info_o)

application(Number_c, Type, Year, Item, Researchers,
Value_applied, Value_granted, Process_no, Status)

where **Number_c** is the center number, and **Info_c**, **Info_o** and **Item** are references to texts on disk, containing information about the center, the organic and the item(ns) refered in the application, respectively. Three special features were used :

- **center name** : this was made an **implicit field** of the data base predicate **center**, and was defined as a reference to text with the form **t(cent, Number_c)**. We used an interface predicate to implement it though the use of an ancillary condition would be more efficient as the field is an **output-only** one and not so often used

- **director name** : for output we built another implicit field of the form **Director_title : Director_name** (where ':' is an infix operator) implemented through an ancillary condition

- **researchers** : this field is either **0** (zero : no researchers involved or unknown), or of the form

N1+N2+... ('+' being another infix operator) where N1, N2, ... are researcher numbers. When retrieving the applications a given researcher is involved in, we want to find fields containing his number. When output is wanted from this field, we do not want numbers but names, so we build a list of references to texts that have the form t(researcher, N). Accordingly, we use two special predicates to access the field, one for when it is input, the other for when it must be output.

We illustrate the consultation program with a small ficticious data base. In the protocol below, user answers follow the prompt ':' and commentary comes between braces.

```
{ the first menu is }

one) all) how)many s)um m)ean g)reatest l)east o)racle  !  bye
: one                                                  { give me one }

a)pplication  c)enter  o)rganic  !  error
: a                                                    { application }

n)umber_c  t)ype  y)ear  i)tem  r)esearchers
  va)pplied  vg)ranted  p)rocess_no s)tatus all)fields ! error
: n 61                                          { no. of  center  61
: y 1980                                          in the year  1980,
: va?                                             the value applied,
: vg?                                             the value granted,
: !                                               and nothing else }

a)nswer  and  andn)ot  or  orn)ot  error
: a                                                      { answer }

value_granted          25
value_applied          25

{ return to the initial situation after the answer }

one) all) how)many s)um m)ean g)reatest l)east o)racle  !  bye
: o

{ the oracle obtains a complete designation  of  one  or  more
  researchers or centers as desired, whose partial designation
  is known }
```

```
r)esearchers  c)enters  !
: c

write in one line only the partial designation you know
and capitalize proper and common names ;
the usual abreviations are allowed if ended with a dot.
: C. de Informatica

identification no. 61
Centro de Informatica Universidade Nova de Lisboa

m)ore  a)nother  !        { here  one  may  request  complete
: m                       designations for the same partial one,
                          or supply another partial designation,
                          or (!) terminate }

unknown

r)esearchers  c)enters  !   { consultation ends and the system
: !                           returns to the initial situation }


{ give me all about an application from center no. 61 }

one) all) how)many s)um m)ean g)reatest l)east o)racle  !  bye
: one

a)pplication  c)enter  o)rganic  !  error
: a

n)umber_c  t)ype  y)ear  i)tem  r)esearchers
  va)pplied  vg)ranted  p)rocess_no s)tatus all)fields ! error
: n 61
: all
: !

a)nswer  and  andn)ot  or  orn)ot  error
: a

number_c          61
type              2
year              1980
item              visit of David Warren
researchers       Luis Moniz Pereira
value_applied     25
value_granted     25
process_no        347
status            ok
```

```
{ how many are the applications ? }

one) all) how)many s)um m)ean g)reatest l)east o)racle  !  bye
: how

a)pplication  c)enter  o)rganic  !  error
: a
      { a ! would return the system to the initial situation }

n)umber_c  t)ype  y)ear  i)tem  r)esearchers
  va)pplied  vg)ranted  p)rocess_no s)tatus all)fields ! error
: !
        { no further specification is intended ;
          ! is given since the specification has ended }

a)nswer  and  andn)ot  or  orn)ot  error
: a

number          21

one) all) how)many s)um m)ean g)reatest l)east o)racle  !  bye
: all           { give me for all }

a)pplication  c)enter  o)rganic  !  error
: c             { centers }

n)umber_c  i)nitials  s)ector  d)istrict  in)fo_c  na)me
                                          all)fields  !  error
: n?            { their number and
: i?              their initials,
: d*              grouping them by district (*) }
: !

a)nswer  and  andn)ot  or  orn)ot  error
: a

initials-number_c by district
* lisboa  :

cipd  76              is  206         spm  7
spb  19               spcv  20        ...

* porto  :

spo  40               cemup  15       demfeup  11
deafeup  44
```

```
{ sive me, for all centers, their name,
  sroupins them by district and by sector }

one) all) how)many s)um m)ean s)reatest l)east o)racle  ! error
: all

a)pplication  c)enter  o)rsanic  !  error
: c

n)umber_c  i)nitials  s)ector  d)istrict  in)fo_c  na)me
                                          all)fields  !  error
: na?
: s*
: d*
: !

a)nswer  and  andn)ot  or  orn)ot  error
: a

name by sector-district
* lisboa  ipsfl  :

Centro de Informatica e Pesquisa para o Desenvolvimento
Instituto de Soldadura
Sociedade Portuguesa de Matematica

* porto  ipsfl  :

Sociedade Portuguesa de Ornitologia

* lisboa  sovernment  :

Direccao-Geral do Saneamento Basico

* lisboa  ensino_inic  :

Centro de Fisica Nuclear da Universidade de Lisboa
Centro de Informatica Universidade Nova de Lisboa

* porto  ensino_inic  :

Centro de Ensenharia Mecanica da Universidade do Porto

* lisboa  ensino  :

Departamento de Estudos Classicos

* porto  ensino  :

Departamento de Ensenharia Mecanica
        da Faculdade de Ensenharia da Universidade do Porto
Departamento de Ensenharia Quimica
        da Faculdade de Ensenharia da Universidade do Porto
```

```
one) all) how)many s)um m)ean g)reatest l)east o)racle  !  bye
: g              { give me the greatest value }

a)pplication  c)enter  o)rganic  !  error
: a              { in an application }

n)umber_c  t)ype  y)ear  i)tem  r)esearchers
  va)pplied  vg)ranted  p)rocess_no s)tatus all)fields ! error
: y 1980         { for the year 1980
: vg?              that was granted }
: !

a)nswer  and  andn)ot  or  orn)ot  error
: a

greatest         value_granted   400


{ give me, for all applications, the item and the center name
  grouped by district }

one) all) how)many s)um m)ean g)reatest l)east o)racle  !  bye
: all

a)pplication  c)enter  o)rganic  !  error
: a

n)umber_c  t)ype  y)ear  i)tem  r)esearchers
  va)pplied  vg)ranted  p)rocess_no s)tatus all)fields ! error
: i?
: !

{ to refer the name and the district one needs to consider the
  center ;  to do so, a conjunction is made  of  the  previous
  partial request with an additional specification }

a)nswer  and  andn)ot  or  orn)ot  error
: and

a)pplication  c)enter  o)rganic  !  error
: c

n)umber_c  i)nitials  s)ector  d)istrict  in)fo_c  na)me
                                          all)fields  !  error
: n=         { it is necessary that the center to which we refer
: na?        now be the same center assumed in the specification
: d*         of the application ; thus the n=  ;   =  allows  in
: !          in general  to  link  among  themselves  successive
             partial specifications }
```

```
a)nswer  and  andn)ot  or  orn)ot  error
: a

item-name by district
* porto   :

Varian type atomic absortion espectrofotometer
Departamento de Ensenharia Quimica
        da Faculdade de Ensenharia da Universidade do Porto

fundins of non-profitable private institution
Sociedade Portusuesa de Ornitolosia

* lisboa   :

fundins of non-profitable private institution
Centro de Informacao e Pesquisa para o Desenvolvimento

visit of Dr. Fullyear to Alcabidexe
Instituto de Soldadura

fundins of non-profitable private institution
Sociedade Portusuesa de Matematica
                ...


{ sive me all about every center not in the district of Lisbon }

one) all) how)many s)um m)ean s)reatest l)east o)racle  !  bye
: all

a)pplication  c)enter  o)rsanic  !  error
: a

n)umber_c  i)nitials  s)ector  d)istrict  in)fo_c  na)me
                                        all)fields  !  error
: all
: !

a)nswer  and  andn)ot  or  orn)ot  error
: andn  { here the partial specification is completed with the
          negation of a  conjuncted  additional  specification
          andn ;  it is also possible  to  continue  with  an
          alternative or or with a negated  alternative  orn }

a)pplication  c)enter  o)rsanic  !  error
: c

n)umber_c  i)nitials  s)ector  d)istrict  in)fo_c  na)me
                                        all)fields  !  error
: n=            { the center is the same }
: d lisboa
: !
```

```
a)nswer  and  andn)ot  or  orn)ot  error
: a

number_c - initials - sector - district - info_c - name

40  spo ipsfl  porto  address1
    Sociedade Portuguesa de Ornitologia

15  cemup ensino_inic  porto  address2
    Centro de Engenharia Mecanica da Universidade do Porto

11  demfeup ensino  porto  address3
    Departamento de Engenharia Mecanica
        da Faculdade de Engenharia da Universidade do Porto

44  deqfeup ensino  porto  address3
    Departamento de Engenharia Quimica
        da Faculdade de Engenharia da Universidade do Porto


{ give me all about each organic in 1980 }

one) all) how)many s)um m)ean g)reatest l)east o)racle  !  bye
: all

a)pplication  c)enter  o)rganic  !  error
: o

n)umber_c  y)ear  dt)itle  dn)ame  i)nfo_o  d)irector
                                        all)fields ! error
: all
: y 1980
: !

a)nswer  and  andn)ot  or  orn)ot  error
: a

number_c - year - info_o - director

11  1980  info1  prof:Vasco Sa
61  1980  info2  prof:Candido Marciano da Silva


one) all) how)many s)um m)ean g)reatest l)east o)racle  !  bye
: how

a)pplication  c)enter  o)rganic  !  error
: error              { "how" was not intended ; the program is

[EXECUTION ABORTED]  automatically restarted  after an error }
```

```
one) all) how)many s)um m)ean g)reatest l)east o)racle  !  bye
: ho
 what ? { ho is not an option }
: o

r)esearchers  c)enters  !
: r

write in one line only the partial designation you know
and capitalize proper and common names ;
the usual abreviations are allowed if ended with a dot.
: Guedes

identification no. 1
Dr Joao Guedes de Carvalho 44

m)ore  a)nother  !
: m

identification no. 2
Dr Rodrigo Guedes de Carvalho 44

m)ore  a)nother  !
: !                        { return to the initial situation }


one) all) how)many s)um m)ean g)reatest l)east o)racle  !  bye
: bye                      { exit from program }

Bye !
```

Generating Consultation Programs

The consultation program described in the last section was designed from scratch, implemented and thoroughly tested in a man-month on a PDP-11/23 with 128KB central memory and 2 floppy-disks (RX02). Soon it was felt that a 'meta-consultation program' was within reach and would be extremely useful. In fact the data base dependent sections of the consultation program were easily set apart, and provision was made to generate these sections from answers cajoled from the user about his data base. In some 6 man-days the generating program in Prolog was finished and tested.

It is our strong conviction that all this was only possible through the use of Prolog.

Generated programs are concise (contain exactly what is needed to implement the features selected by the user) and to

some extent are protected from errors (e.g. duplicate names). Obviously, the user must provide any Prolog subprogram or special access predicate alluded to when generating the consultation program.

Below, we present a sample session with the generating program, regarding the consultation program of the previous section. Again we use ':' to prompt the user.

```
Hello !
In case you have any doubt type '?' for help.

output file ? : ?
Please give the name of the file where the consultation
        program is going to be written to.
output file ? : facc

data base file ? : nucl

password - "no" if none ? : xxx

do you need integers greater than 16383 ? : yeah
acceptable answers - [yes,no]
do you need integers greater than 16383 ? : yes

Now, some questions concerning quantifiers and subprograms
        called from the 1st menu of the access program.

do you need arithmetic ? : yes
Which of the following do you need -

sum ? : yes
mean ? : yes
greatest value ? : yes
least value ? : yes
do you need other functions ? : no

are there references to texts in the data base ? : yes
do you want to include an 'oracle' ? : ?
The oracle is a subprogram that finds complete designations
        from partial ones. The complete designations should be
        grouped into different files according to their mean-
        ings - for instance, names of people, organizations.
do you want to include an 'oracle' ? : yes

mnemonic for group of designations ? : c
rest of name ? : enters
file containing this group ? : cent

more groups ? : yes
```

```
mnemonic for group of designations ? : r
rest of name ? : esearchers
file containing this group ? : researcher

more groups ? : no

do you want to make calls to other subprograms ? : no

Questions concerning data base predicates, their fields and
        access to them -

mnemonic for data base predicate ? : c
rest of name ? : enter

predicate name ? : dbase
predicate name already in use ; try another
predicate name ? : center

no. of explicit fields for this predicate ? : 5

do you want an interface predicate for this db one ? : yes

name of the interface predicate ? : cent

no. of implicit fields created by this interface ? : 1

an implicit field is V6
what is the Prolog condition for it - do not use blanks ! - ?
: V6=t(cent,V1)
V6=t(cent,V1) ok ? : yes

mnemonic for predicate field ? : n
rest of name ? : umber_c
heading for output ? : number_c
a normal field ? : ?
A normal field is a  both-way  field  that  needs  no  special
        predicates to be accessed.
a normal field ? : yes

mnemonic for predicate field ? : n
field mnemonic already in use ; try another
mnemonic for predicate field ? : i
rest of name ? : nitials
heading for output ? : initials
a normal field ? : yes

mnemonic for predicate field ? : s
rest of name ? : ector
heading for output ? : sector
a normal field ? : yes

mnemonic for predicate field ? : d
rest of name ? : istrict
```

```
headings for output ? : district
a normal field ? : no
a both-way field : no
an output only field ? : yes
access by special predicate ? : no

mnemonic for predicate field ? : in
rest of name ? : fo_c
headings for output ? : info_c
a normal field ? : yes

mnemonic for predicate field ? : na
rest of name ? : me
headings for output ? : name_c
a normal field ? : no
a both-way field : no
an output only field ? : yes
access by special predicate ? : no

any implicit field created by ancillary conditions ? : no

more data base predicates ? : yes

mnemonic for data base predicate ? : o
rest of name ? : rganic
predicate name ? : organic

no. of explicit fields for this predicate ? : 5

do you want an interface predicate for this db one ? : no

mnemonic for predicate field ? : n
rest of name ? : umber_c
headings for output ? : number_c
a normal field ? : yes

mnemonic for predicate field ? : y
rest of name ? : ear
headings for output ? : year
a normal field ? : yes

mnemonic for predicate field ? : dt
rest of name ? : itle
headings for output ? : *
a normal field ? : no
a both-way field : no
an output only field ? : no
access by special predicate ? : no

mnemonic for predicate field ? : dn
rest of name ? : ame
headings for output ? : *
a normal field ? : no
```

```
a both-way field ? : no
an output only field ? : no
access by special predicate ? : no

mnemonic for predicate field ? : i
rest of name ? : nfo_o
heading for output ? : info_o
a normal field ? : no
a both-way field ? : no
output only field ? : yes
access by special predicate ? : no

any implicit field created by ancillary conditions ? : yes

mnemonic for field created by an ancillary condition ? : d
rest of name ? : irector
heading for output ? : director

result is V6
what is the ancillary condition - do not use blanks ! - ?
: V6=V3:V4
V6=V3:V4 ok ? : yes

more ancillary conditions ? : no

more data base predicates ? : yes

mnemonic for data base predicate ? : a
rest of name ? : pplication
predicate name ? : application

no. of explicit fields for this predicate ? : 9

do you want an interface predicate for this db one ? : no

mnemonic for predicate field ? : n
rest of name ? : umber_c
heading for output ? : number_c
a normal field ? : yes

mnemonic for predicate field ? : t
rest of name ? : ype
heading for output ? : type
a normal field ? : yes

mnemonic for predicate field ? : i
rest of name ? : tem
heading for output ? : item
a normal field ? : no
a both-way field ? : no
an output only field ? : yes
access by special predicate ? : no
```

```
mnemonic for predicate field ? : r
rest of name ? : esearchers
heading for output ? : researchers
a normal field ? : no
a both-way field ? : yes
access by special predicate ? : yes
when a value is input ? : yes

if input value is V and field value is V4
what is the Prolog condition - do not use blanks ! - ?
: res_in(V,V4)
res_in(V,V4) ok ? : yes

and when a value is output ? : yes

if output value is X4 and field value is V4
what is the Prolog condition - do not use blanks ! - ?
: res_names(V4,X4)
res_names(V4,X4) ok ? : yes

mnemonic for predicate field ? : va
rest of name ? : pplied
heading for output ? : value_applied
a normal field ? : yes

mnemonic for predicate field ? : vg
rest of name ? : ranted
heading for output ? : value_granted
a normal field ? : yes

mnemonic for predicate field ? : p
rest of name ? : rocess_no
heading for output ? : process_no
a normal field ? : yes

mnemonic for predicate field ? : s
rest of name ? : tatus
heading for output ? : status
a normal field ? : yes

any implicit field created by ancillary conditions ? : no

more data base predicates ? : no

Your consultation program is in file  facc
Don't forget appending to it the definitions of the special access
        predicates you mentioned here.

Bye !
```

Conclusions

Prolog is an excellent unrivaled language for this type of application, but there is still room for improvement through research. In particular, large data bases require more indexing facilities, and multi-user access with on-line updating poses special protection problems. Imposing integrity constraints is also beginning to be explored in the context of logic programming. Query planning and more intelligent access mechanisms in general are also in order.

References

L.M. Pereira, M. Filgueiras

82 Manual de Utilizacao da Base de Dados FACC
Departamento de Informatica
Universidade Nova de Lisboa

W. Clocksin, C. Mellish

81 Programming in Prolog
Springer Verlag

MODELLING HUMAN-COMPUTER INTERACTIONS

IN A FRIENDLY INTERFACE

Patrick SAINT-DIZIER
UNIVERSITE DE RENNES I
I.R.I.S.A.
Campus de beaulieu
35042 RENNES FRANCE

A B S T R A C T

This paper describes a way of building an intelligent interface between a human and a computer. We first examine the main characteristics of such a system from three points of view: the user, the expert in the definition of applications and the computer scientist. Our job is to find an appropriate formalism that can describe the different aspects of our system: natural language understanding, knowledge representation, explanation and control mechanisms, plan generation, relational databases and meta-knowledge representation. Finally, we conclude by pointing out some remaining problems that will require additionnal basic research.

The job described here has been implemented in PROLOG computing language on the CII HB 68 of the IRISA. Our assistant interfaces an application, called Cigare, whose role is to help people in scheduling meetings.

Keywords: Office automation, knowledge representation, natural language, relational databases and cognitive modeling.

1_ INTRODUCTION :

The close interactions between the fields of computer science and cognitive psychology have given rise to what is generally known as human information processing models of cognitive processes. In this paper, we describe an intelligent interface between a human and a specialized computer (such as: text-editor, interactive query system, electronic-mail, ...). We call this interface a user assistant. Its role is to be an expert of the functions of the application to which it is connected and to provide a friendly environment to people who wish to use this application. In this job, we look on our assistant from three points of view:

-the user's point of view. He uses the application via the interface. Very often, he is not a computer professional.

-the computer scientist's point of view. Its role is to build a man-machine interface adaptable to various kinds of applications.

-the expert's point of view. An expert is a specialist in the definition of applications. His task is to specify to the assistant the different parameters of the application he wants to interface.

In the next section, our purpose is to gather some of the most interesting characteristics a human-computer interface must have, from the user's point of view. In section 3, we take up the expert position and we describe what structures are necessary so as to enable the expert to describe properly the different aspects of the application he wants to interface. Finally, in section 4, we describe the main problems the computer scientist is confronted to.

This job is an attempt to formalize and to solve some of the problems we have met in our investigations. We don't claim to solve all the difficulties and we think that a lot of basic research remains to be done.

2_ A FRIENDLY INTERFACE FOR USERS

With the help of psychologists and application builders, we have brought out some of the main properties an intelligent interface must have [11]:

2.1_ Understanding the User's Requests:

Interacting with a computer in a "natural" way is much more perceived as "user friendly". In fact, it will be always necessary for users to make the effort of learning what a system is capable of doing, but natural language would minimize the efforts of learning how to make the system do it. In addition, natural language allows a casual user to use more advanced capabilities of a system without knowing the exact commands. But, as yet, due to the vast amount of information to store, the domain of the discourse has to be highly restricted. The assistant has, in fact, to know far more than the syntactic rules that allow translating natural language into a formal representation (cf. section 4). In order to avoid frustating the user, the domain of the discourse has to be carefully studied for the assistant can understand immediately most of the inputs and the rest after one or two rephrasings.

In other respects, some experiences have shown that when users have some difficulties to express a request (especially a request for help), the best solution is that the assistant provides them with a menu driven dialog. However, this menu driven dialog has to be carefully studied in order to take into account the main problems a user can meet.

When the user expresses himself in a "natural" way, most of his requests will not correspond exactly to the input forms of the service actions. The user assistant must therefore have the capability of

mapping a given request into the appropriate commands of the service that fulfil that request.

It is also important that the assistant generates a paraphrase of what it has understood in a request and that it waits for an acknowledgement from its user before it sends a command to the application.

2.2_ Responding Intelligently to the User's Inquiries:

In order to be really friendly and helpful, the assistant must be able to answer questions about:

-informationsthe user has already submitted,

-the current state of the application;

-the linguistic competence of the assistant (words or grammatical structures it knows). The user may ask for exemples of sentences the assistant can parse.

-how to perform a given task,

-which tasks the user may perform at that precise moment,

-why he can't perform a task,

-why and how the assistant has made some deductions (ex: how it has solved unknown references).

In the answer of the assistant, a recall of the question has to be mentionned so as to ensure the user that the assistant has understood properly his question.

2.3_ Detecting User's Failures and Making Explicit the System Limits:

Both the user and the assistant may fail for various reasons. We differentiate two types of failures: input and model failures. Input failures are due on the one hand to grammatical and semantic mistakes from the user and on the other hand to linguistic structures the assistant doesn't know and to misperception of a word when using vocal terminals. Model failures are due to the user's ignorance or bad understanding of some aspects of the application.

The assistant has to detect these failures and to provide explanations to its user. Concerning input failures, the assistant has to point out unknown words, semantic incompatibilities and sentence structures it can't parse. When it meets an unknown word in a sentence, it must try to deduce the meaning of this word from the remainder of the sentence. If it succeeds, it has then to ask the user for a confirmation. Concerning model failures, it must point out false presuppositions, incomplete requests and unknown actions to the application. It must be really informative (but not too talkative !), showing clearly and explicitly why it cannot accept a request. It must provide explanations, exemples and alternatives or restatements.

2.4_ Acquiring knowledge :

The user assistant is particular to a user. Consequently, it must be adaptable to his sensibility and habits. It must be able to learn some new information, so as:

-to increase its linguistic competence. Our assistant can learn new synonyms of words that it already knows. The user as just to declare: "X is a synonym of Y". We think that the acquisition of new words and new grammatical structures has to be done by a human expert in linguistics because most of the users have not the required competence to perform this task.

-to take into account behaviour specifications, in order to avoid disturbance to its user. The user may give instructions to its assistant, such as "My meetings always take place in room no 210.". They play the role of default options.

3_ ROLE OF THE EXPERT

The assistant, application independant, is built by a computer scientist. To interface a given application, some parameters of the assistant have then to be instantiated. This is the role of the expert. The two main classes of parameters are the linguistic parameters and the parameters that describe the functions of the application. The specification of these parameters is done via a specific language.

3.1_ The Linguistic Parameters:

In the previous section, we have explained that the domain of the discourse has to be restricted. Consequently, it is not possible to store in our assistant, once for ever, all the vocabulary of a given language. That means that the expert will have to find , for each application to interface, all the required vocabulary so as to allow the user to express himself in natural language with sufficiently various expressions. Some words, such as articles and prepositions are common to all the applications but most of the words are application dependant. In order to limit the job of the user, it is useful to implement in the assistant, once for ever, a set of rules that describe the various morphologies (plural, feminine, conjugation forms ...) of any given word. So, the expert has only to specify the infinitive form of verbs, the masculin singular of adjectives, etc... For each of these words, the expert has to give:

-the syntactic category of the word (noun, verb ...),

-if this word accepts complements.

This last point leads us to introduce semantic features so as to enable the expert to precise what kind of complement is acceptable. As we are concerned by a small subset of natural language, it is possible to define semantic categories in a finite number and to include each word in, at least, one of these categories. We think that these categories are limited to the set of categories of objects on which the application operates (human, time, place ...). Finally, the structure of a lexical

item is composed of :

-a word,

-a syntactic feature (noun, verb,...),

-a semantic feature, linked to the word itself (except for verbs where this feature is the feature of an acceptable subject),

-a list of semantic features of acceptable complements, with the preposition that introduce them.

Exemple:

WORD(assembly,noun,meeting,(of,human).(of,place).nil).

The rules that describe the grammatical structures are application independant. The main rules are implemented in the assistant once for ever. However, we think it is important to allow the expert to add new grammatical structures and descriptions of idiomatic expressions. This can be done via a specific language [6].

3.2_ The Description Of The Application:

The expert describes a model of the application to the assistant. The first goal of this description is to make the assistant "understand" the kind of request the user has submitted. The second role is to enable the assistant to help the user. The word "understand" means, here, to find the exact meaning of a request with regard to the functions of the application. It also means to verify if this request is possible considering the previous actions the user has performed and the data transmitted by the application.

To model an application, the first task is to decompose it into basic actions. An action will be identified by a set of significant patterns to find in the output form produced from the user's request. Next, it is necessary to describe the conditions under which an action may be performed. These conditions express that, previously, some actions must have (or must not have) been performed by the user and (or) some information must have (or must not have) been transmitted by the application. These information are the result of the user's actions or the result of other user's actions in the case of multi-users applications. Finally, for each action, an exhaustive list of tasks the assistant has to do is described in terms of information to add to a contextual database and commands to send to the application.A rule that describes a basic action of an application is of the form :

RULE(<identification>,
<list of patterns to find in the request>,
<conditions >,
<information to add to the contextual database>.
<commands to send to the application>).

4_ ARCHITECTURE OF THE SYSTEM : THE COMPUTER SCIENTIST POINT OF VIEW.

The job of the computer scientist is to build a system that takes

into account both the different parameters specified by the expert and the human-computer interactions constraints we have point out in section 2. In this section, we first examine the linguistic component and next see how the request is interpreted. Finally, we explain how the assistant can provide help to its user.

We first assume that the human and the machine interact in a way that can be described by a production system. Indeed, we think that the formalism of production systems and logic is a good formalism that has been really successful in providing insights in both theoretical and practical aspects of computing science. Then, we have to specify:

-A general structure to describe the rules of this production system,

-the process by which rules are selected for execution, and how to express it,

-the structure of the information utilized by the rules,

-how the information reflects the current state of the knowledge on which the system operates,

-the operations on the rules (modifications, ...).

Figure 1 shows the overall structure of a user assistant:

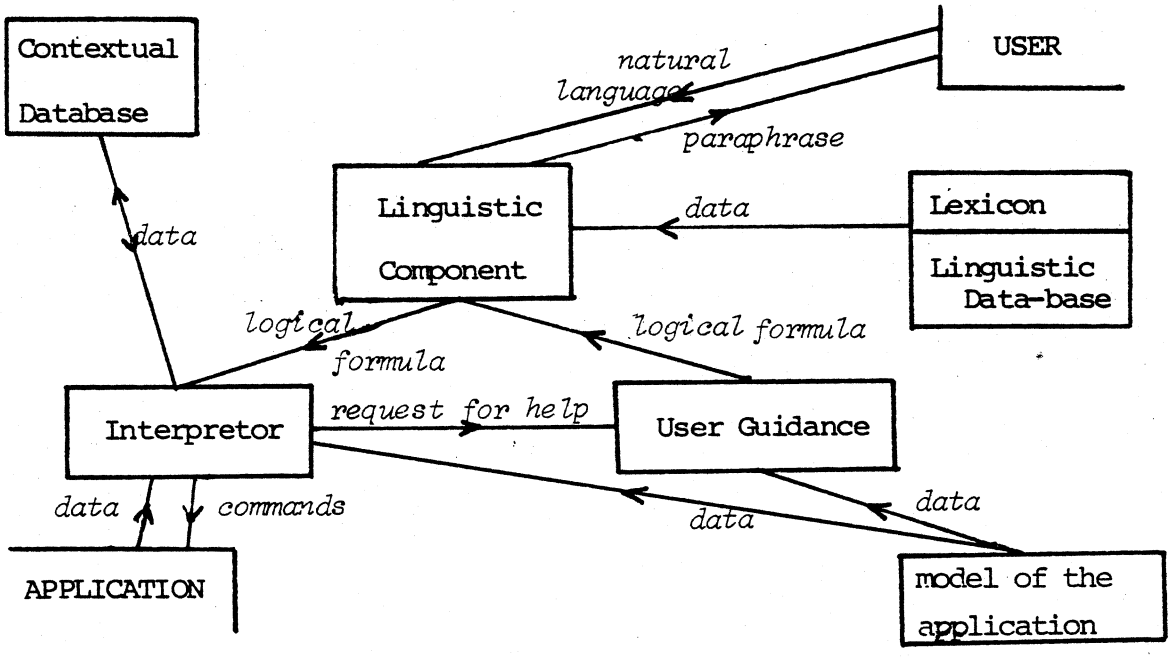


fig. 1

4.1 The Linguistic Component:

The goal of the linguistic component is to generate a formal representation of the meaning of the natural language sentences of the user. Actually, there exist many various ways to represent formally a

natural language sentence. We think that the formalism adopted and described by [6], [8] and [17] is very well adapted to our problem. In this formalism, the study of determiners have been looked at indetail so as to refine the range of quantifiers. In addition, some tools have been added so as to represent better some structures such as questions begining by : Why, How many, How much and the expression of time. This representation is in higher level logical form. For instance, the question

"Who are the participants of the meeting A ?"
has the following formal representation :
QUESTION(SET-OF(x),. meeting (A).participant-of(x,A))

Our parser is composed of two entities: a lexicon and a set of rules that describe the grammatical structures of french. We think that it is important that these rules may also be applied backward, for sentence synthesis. In fact, a lot of job remains to be done about this problem. The main problems we are confronted to about sentence synthesis is that we must specify all the syntactic constraints, so as to have a correct output, and to ensure ourselves that all the information the logical form contains has been synthetised in a correct way.

In our system, a sentence is parsed by a grammar where:

(1) The axioms are a finite set of modes : declarative, imperative....

(2) The non-terminal symbols (SN, SV, Verb ...) have the general form:
X(<syntactic features>,<semantic feature>,<formal representation>)

The syntactic features (gender & number) and the semantic feature (human, place, ...) are the features that result of the parse of the sentence substructure represented by X.

(3) The terminal symbols, that are the lexical items.

(4) The rules, that have the form:
$X(l(SY_1,\ldots SY_i),g(SE_1,\ldots SE_i),h(F_1,\ldots F_i)) \longrightarrow$
$Y_1(SY_1,SE_1,F_1) \ldots\ldots Y_i(SY_i,SE_i,F_i)$

are applicable iff $s1(SY_1,\ldots SY_i)$ and $s2(SE_1,\ldots SE_i)$ are true. Where:

- SY stands for the syntactic features,
- SE stands for the semantic feature,
- F for the formal representation of a substructure of a sentence.

s1 and s2 are functions that express conditions, such as pattern-matching, between features that come from each non terminal symbol on the right part on the rule. l and g are functions that describe how to build the syntactic and semantic features of the non terminal symbol on the left part of the rule from the features on the right part of that rule. f describes how to build the formal representation of the sub-expression parsed by the rule. The main problem is that the meaning of a complex expression has to depend only on the meaning of its subexpressions. Every well formed subexpression is then considered as a unit of meaning that can be integrated in a larger

expression.

Finally, the lexicon and the rules of the parser are considered as a database so as to enable the system to give information about its linguistic competence (cf. the nice job referred in [14]). This structure also allows us to implement procedures that detect, in a simple way, grammatical mistakes and semantic inconsistencies.

4.2_ The contextual database :

We think that it is important that a request don't be treated as an isolated event. A context is built up so as the repeated exchanges between the user and his machine may be considered as approaching a simple but real conversation. An image of all the exchanges between the user and his assistant and between the assistant and the application is stored in a contextual database. The contextual database is local to a user and is composed of a list of facts that represent:

(1) The information the user has transmitted to the application via his assistant. This can be looked as an historical database.

(2) Informations about the state of the dialog between the user and his interface (what the user knows and what he is talking about).

(3) The information the application has transmitted.

(4) Some behaviour specifications, given by the user.

The contextual database plays the role of a short term memory. All the facts are represented in the same way :

FACT(<kind of fact>,<information>,<source of the information>).

The argument "source of the information" allows the assistant to explain, at any time, the origin of the information (inheritance or deduction, default option, the user's request).

4.3_ Interpretation Of The User's Request:

We have examine in section 3 the structure of the rules that describe an application. In our job, a rule is of the form:

Rule(<name>,C1,C2,L,T).

Where:

-C1 is a set of conditions on the existence of some facts, stored in the contextual database,

-C2 is a set of patterns to find in the logical form produced from the user's request. C2 allows an efficient preselection of rules and is a tool for user guidance (cf. 4.4),

-L is a list of information to collect in the logical form . Some control procedures, linked to the informations, are described here (cf. example),

-T is a list of actions to perform (commands to send to the application and information to store in the contextual database).

Let's look at an example :

```
RULE(positive_answer_to_an_invitation,
    facts(exist(meeting).to_be_invited(user,meeting).nil),
    patterns_to_find(agreement_to_come.nil),
```

```
to_collect(dates(day<=31 and 8<=time<=20).nil),
actions(addCD(positive_answer(user).dates(<dates>).nil).
smAPP(positive_answer(user,<dates>).nil).nil)).
```

addCD stands for "add to the contextual database"
smAPP stands for "send commands to the application"

After the parsing process of the user's request, the interpreter evaluates the C1 and C2 of each rule. Thus, a preselection of (one or more) applicable rules is done. Then, the interpretor asks the user for a confirmation of its understanding. If it is correct, L and T of these rules are executed. If it is not correct, the user has to say which rule is applicable, if any. The user may also ask for help.(cf. user guidance module).

4.4 User Guidance:

One of our main principles is to never left the user to himself. The assistant must be able to help the user at his request or when it detects some failures. User guidance is a very vast problem, let's look at some aspects of it:

(1) What is the linguistic corpus necessary to express requests for help? Is natural language well adapted?

(2) When the assistant is not able to answer a question, how to make it express the reasons why it cannot (instead of the laconic expression "I don't know")? How to make it propose alternatives?

(3) How to help the user to plan his work?

(4) How to learn to novices the main functions of an application?

(5) How to be really very informative, without excess? For instance, is it possible to define different levels of information?

Despite the current interest in user guidance, we think that the design of a helpful and informative interface remains to be done. However, some very interesting and valuable results have been obtained in some works such as : [2], [14], [17], ... In our job, the formalism of the rules that describe an application allows the assistant to provide some explanations:

- When the user doesn't know which actions he may perform at a precise moment, it is from the context and through the evaluation of C1 of all the rules that the assistant gives him the list of the actions he is allowed to perform.
- If the user doesn't know how to perform a given action , the description of the L of that rule gives him the amount of knowledge required to perform this action.
- The evaluation of the C1 and the addCD of T of all the rules allows the assistant to generate plans [9] and to propose to the user various chains of actions (or subgoals) to reach the requested goal. This is a way to learn to novices how to use the application.
- When the user wants to perform an action that is not allowed, the evaluation and the description of the C1 of that rules gives the reasons why this action is not possible, and under which conditions it would be possible.

Actually, the user is guided by a menu-driven language to formulate his request for help. Requests for help in natural language are, indeed , very difficult to express and to be interpreted.

5_ CONCLUSION .

We have presented here a human-computer interface which is to be used by a large and casual public. We have check off what must be the main properties of such an interface. The formalism adopted here, based upon logic, reveals itself to be quite robust and general. This interface has now to be tested by users.

However, a lot of job remains to be done, especially in the following areas:

*How to respond more intelligently to incorrect inputs and to questions about the knowledge.

*How to built an expert that is able of reasoning on incomplete knowledge.

*How to process some linguistic problems such as fuzzy expressions.

*How to build an efficient "expert" to manage knowledge acquisition.

The job we have presented here has been implemented in PROLOG on the CII-HB 68 of the IRISA. The implementation has lasted the equivalent of 14 monthes for one person, but some work remains to be done to increase the performances. The application that our assistant interfaces is called CIGARE, its role is to help people in scheduling meetings. We also intend to connect this interface to a text editor.

REFERENCES

[1] BOBROW D.G., KAPLAN R.M., KAY M., WINOGRAD T. GUS: a frame driven-dialog system. A.I. Vol 8 no 1 1977.

[2] KAPLAN S.J. Cooperative Responses from a portable Natural Language Query System. Artificial Intelligence n° 19 , 1982.

[3] DOYLE J. A glimpse on truth maintenance. Proc. of the IJCAI 79 .

[4] GROSS M. Methodes en syntaxe. Hermann. PARIS 1975.

[5] HIRSHMAN L., PUDER K Restriction Grammars in PROLOG. Logic programming conference, Marseille 1982.

[6] MCCORD M.C. Using slots and modifiers in logical grammars for natural language. Technical report no 69-80. Univ. of Kentucky, Lexington, 1982.

[7] MINSKY M. A framework for representing knowledge, in P. Winston(ed).

[8] MOORE R.C. Problems in logical form. SRI project report . April 1981.

[9] NILSSON N. Principles of artificial intelligence. Tioga Pub. Co 1980.

[10] NORMIER B. Le Systeme SAPHIR. ERLI. 1982.

[11] QUINIOU R., SAINT-DIZIER P. Man-machine interface for large public applications. IEEE Seminar Zurich March 1982.

[12] PEREIRA F. Extraposition Grammars. Logic programming workshop. Debrecen, Hungary 1980.

[13] REITER R., NASH-WEBER B. Anaphora and logical form: on formal meaning representation for natural language. 5th IJCAI.

[14] SABATIER P., PEREIRA L.M. ORBI: an expert system for environmental ressource evaluation through natural language. Logic Programming Conference, Marseille, 14-17 september 1982.
[15] SAINT-DIZIER P. Some aspects of knowledge representation in an intelligent Man-Machine interface. The Mind and Machine Congress, Middlesex polytechnic, London, April 1983.
[16] SHANK R.C., ABELSON R.P. Scripts, plans, goals and understanding. Lawrence Erlbaum Press, Hillsdale N.J. 1977.
[17] WARREN , PEREIRA F.: An efficient easily adaptable system for interpreting natural language. Research report 1981.
[18] BESNARD P., QUINIOU R., QUINTON P., SAINT-DIZIER P., TRILLING L. Dialogue et representation des information dans un systeme de messagerie intelligent. Research report no IRISA 185. January 1983.

A Kernel for a General Natural Language Interface

Miguel Filgueiras

Nucleo de Inteligencia Artificial

Departamento de Informatica
Universidade Nova de Lisboa
Quinta da Torre
2825 Monte da Caparica
PORTUGAL

Abstract

A description is given of the main ideas used in the design of SPIRAL, a kernel for a natural language interface aimed at generality in linguistic ability and domain portability.

Though Prolog is used to implement the interface, syntactic analysis is not performed via metamorphosis, definite-clause, or extraposition grammar formalisms, but rather by means of a 3-level bottom-up extensible parser making use of rewrite rules. The application of each of these rules is controled by a module capable of embodying non-syntactic knowledge.

Syntactic and semantic analyses are separately done, but semantic tests are embedded in the parser resulting in a substantial decrease of ambiguity. The application dependent parts of the semantic analyser constitute a separate module. To make it easy to adapt the interface to new applications, a set of predicates is provided to help in the definition of that module.

Introduction

> "... reality may avoid the obligation to be interesting, but (...) hypothesis may not."
>
> Death and the Compass, J. L. Borges

Results from research on natural language understanding systems made during the last 15 years, either implicitly (by failing to meet certain requirements), or explicitly, point out the need for world knowledge, inference, context analysis and the like when trying to analyse a natural language sentence (this need is more acute when phenomena such as anaphora (reference problem) is dealt with [G. Hirst 81]).

One of the main problems with the logic grammar formalisms proposed so far (metamorphosis grammars [A. Colmerauer 75,78], definite-clause grammars [F. Pereira, D. Warren 80], and extraposition grammars [F. Pereira 81]), as well as with their concrete applications (from [R. Pasero 73] to [F. Pereira 83]), is that no provision is made to check each syntactic analysis step for consistency with respect to meaning. In this sense, syntactic analysis is carried out blindly. Introduction of tests in the grammar rules, typification [V. Dahl 77] and slot-filler based approaches [M. McCord 80,81], [F. Pereira 83], are incipient steps toward the use of non-syntactic knowledge to guide parsing. But in present day systems, whenever such knowledge is used it must be interspersed within the grammar rules and there is no neat separation at this level between the syntactic and the non-syntactic modules - even if semantic analysis is performed after syntactic analysis.

In what follows I present the main ideas underlying **SPIRAL**, an open kernel for a general natural language interface that gives an answer to the above criticism and simultaneously tries to keep a high degree of portability between applications.

In **SPIRAL** a 3-level parser is used to perform syntactic analysis. The second and third levels are executed in an interleaved fashion so that the third level checks second level results on the fly. The non-syntactic knowledge the third level has can easily be extended to include criteria based on knowledge from discourse context, world knowledge,

inference, and so forth. This way it is possible to have a desirable interaction between two highly modular devices, one working on the syntactic features and being controlled by the other which uses more complex forms of knowledge.

Syntactic and semantic analyses are separately done, but semantic tests are embedded in the parser resulting in a substantial decrease of ambiguity. The application dependent parts of the semantic analyser constitute a separate module. To make it easy to adapt the interface to new applications, a set of predicates is provided to help in the definition of that module.

Syntactic Analysis

"I state ; you, if you wish, refute."

The Aristos J. Fowles

A first point of divergence from the metamorphosis, definite-clause and extraposition grammar formalisms (referred to as '**logic grammars**' in what follows) is the parsing strategy. The **SPIRAL** parser makes use of a bottom-up technique better suited to accept external guidance and to analyse elliptic sentences and all forms of extraposition (I have no intention of entering the old and tired top-down versus bottom-up controversy - please cf. the quotation above - ; though many people tend to admit that the former is more efficient than the latter, this is **false** at least for (unbiased) context-free grammars [M. Kay 80]). The stress put on the two linguistic phenomena above (ellipsis and extraposition) follows from the purpose of not restricting 'ab ovo' the interface capabilities, and also from the relatively high frequency of such forms in Portuguese, the language actually analysed by **SPIRAL**.

While rules of a logic grammar constitute an indivisible program working on normally 3 types of data (surface representations, non-terminals and syntactic structures), **SPIRAL** is stratified into levels according to the functions performed and the kinds of data dealt with.

Rewrite rules in **SPIRAL** are in some extent similar to the rules in logic grammars. Obviously they occur in inverted forms, in accordance with the bottom-up parsing strategy - while in a logic grammar we have, for instance,

a -> b1, b2 ..., bn

in SPIRAL the same rule will appear as

b1, b2 ..., bn -> a

There is no distinction between terminals and non-terminals. A sentence is represented by the list of the lexical representations for its words, and the lexical representations can have Prolog variables to hold information for future use. Lists of lexical representations are what actually appears on both sides of the rewrite rules. Hence there are no restrictions on a rule's right-hand side, in contradistinction with logic grammars' left-hand sides.

Besides the lexical one, two other representation forms are used : one for what I call meaning-cells (**m-cell**, for short), and another for phrase structures built from them.

A m-cell tries to represent any contiguous words group that is meaningful on its own when isolated from the rest of the sentence. m-cells may contain other m-cells and be conjoined to give a m-cell. Some of the m-cell types **SPIRAL** currently works with are :

- noun phrase
- verb
- complement (an adjective group, a prepositional phrase, or an adverb)
- subphrase (relative clause)
- wh-question

For instance, in the sentence

'The system uses techniques to encode a more general model that are very efficient'

there are the m-cells

- the system
- uses
- techniques
- to encode a more general model
- that are very efficient

the last two of them containing

- encode
- a more general model
- are
- very efficient

Phrase structures are represented by a 3-place functor whose three arguments in a given instant describe of a phrase

- its main m-cells (either a verb, or noun-phrases - verbs are envisaged as phrase 'functors'),

- complements that await attachement to nouns or verbs (this simplifies the treatment of extraposition),

- subphrases found so far.

We can now examine how the **SPIRAL** parser works. On a first level of processing words are conflated whenever possible ; information from deleted words instantiate variables that occur on the lexical representations of the remaining words. This is a deterministic pass and results from applying rewrite rules like the following (in Edinburgh syntax, with '->' as infix operator),

```
[ determiner(Quant,Agr), noun(N,Quant,Agr) | R ]

        ->   [ noun(N,Quant,Agr) | R ]  .
```

This rule states that a determiner followed by a noun is deleted if both have the same agreement. Moreover, the quantification expressed by the determiner is saved in the lexical noun representation. This particular rule is a Prolog unit clause but some other rules have a clause body to test their applicability. After the first level, a second level analyses word groups to obtain m-cells. This is done by applying recursive rewrite rules with the following format :

```
List_of_lex_reprs_1  -->

            List_of_lex_reprs_2  -  M_cell   :- ...
```

where '-->' and '-' are infix operators. Such a rule means that **M_cell** is the result of analysing the first list of lexical representations, the second one being what is remnant. Like for the first level rules, clause bodies may impose conditions on rule application.

Each m-cell extracted by the second level is embedded into the current phrase structure by a third level of processing to produce a new phrase structure. Each clause head in the third level has the format

```
M_cell  +  Phrase_str_1   --->   Phrase_str_2
```

where '+' and '--->' are infix operators - its meaning is obvious. A monitor is used to control the second and third

levels forcing their interleaved execution. This monitor is defined by the following two clauses

```
mon( L0, P0, Ln, Pn )  :-   L0  -->  L1 - M  ,
                            M  +  P0  --->  P1 ,
                            mon( L1, P1, Ln, Pn ).

mon( L, P, L, P ).
```

So, whenever the third level fails by finding out that a m-cell is extraneous to the current phrase structure, backtracking to the second level takes place. In this situation, either an alternative analysis exists, or the monitor stops producing the phrase structure built so far and the rest of the sentence that remains to be analysed. A second level clause body may include a call to the monitor forcing a recursive analysis to be performed.

The first and second levels are purely syntactic, though the latter uses semantic tests to ensure correct attachment of complements to nouns. Both work by applying rewrite rules from two distinct sets comprising, respectively, about 10 and 25 rules. The third level must decide on whether a m-cell can or cannot be added to the current phrase structure. This important function, that imposes a check on each syntactic analysis step, is based, for the time being, on criteria concerning the phrase structure and some knowledge about complements and verb arguments (nouns are typed and for verbs a slot-filler approach is used, as in [V. Dahl 77], [M. McCord 80,81]). Those criteria can easily be extended to more sophisticated ones based on knowledge from discourse context, world knowledge, inference, and so forth.

This way it is possible to have a desirable interaction between two highly modular devices, one working on the syntactic features and dealing with lexical representations, the other using more complex forms of knowledge to build phrase structures.

In summary, the characteristics of these 3 levels in SPIRAL are as follows :

1st level - has some 10 rewrite rules transforming a list of lexical entries into another such list.

2nd level - has some 25 recursive rewrite rules that from a list of lexical entries produce one m-cell and remaining list ; each m-cell is passed to the 3rd level (as soon as produced) and if not accepted, alternative rules (if any) are applied ; otherwise, the processor stops giving as result the phrase structure built so far (if any), and the remnant list.

3rd level - builds the phrase structure from the m-cells extracted by the 2nd level controlling it by accepting or rejecting m-cells ; the 3rd level is also responsible for the treatment of extraposition, passivization, and composite nouns (like '**the dog, the cat and the mouse**').

When the end of the sentence is reached, another **SPIRAL** module is launched to check the phrase structure built for the sentence and to carry on with ellipsis analysis if needed. At present only a few incipient anaphoric forms are analysed by **SPIRAL** and by methods not completely adequate. Personal and possessive pronouns are solved by searching a noun list (built during the lexical analysis) and selecting a noun from it ; some kinds of ellipsis are solved by the introduction and dereferencing of a pronoun, and some others are treated by comparing phrase structures. The general philosophy prescribed in [G. Hirst 81] will sooner or later be adopted in **SPIRAL**. Nevertheless, the semantic tests used in the second and third levels, together with the slot-filler approach, provide a lot of information extremely useful in solving ambiguities. This fact allows for present methods to work well in many instances. A similar situation is encountered in the case-grammar approach, qualified in [G. Hirst 81] as "... **a firm base for anaphora resolution**", though only information from cases is used.

To help fix ideas, two (**simplified**) examples of sentence analysis follow - the two sentences are from [F. Pereira 81,83]. The functor ps(_,_,_) is used for phrase structures (see above for a description of its arguments), and * and ___/ are used to mark, respectively, a failure at the third level, and the words activating a second level rewrite rule. Important information bound to variables on the lexical representations of nouns or verbs is shown informally following them and within parentheses (e.g., **mouse(the)** represents the noun '**mouse**' containing the information from the determiner '**the**') ; in verbs, subject always precedes direct object. Numbers within braces denote comments to be found after each figure.

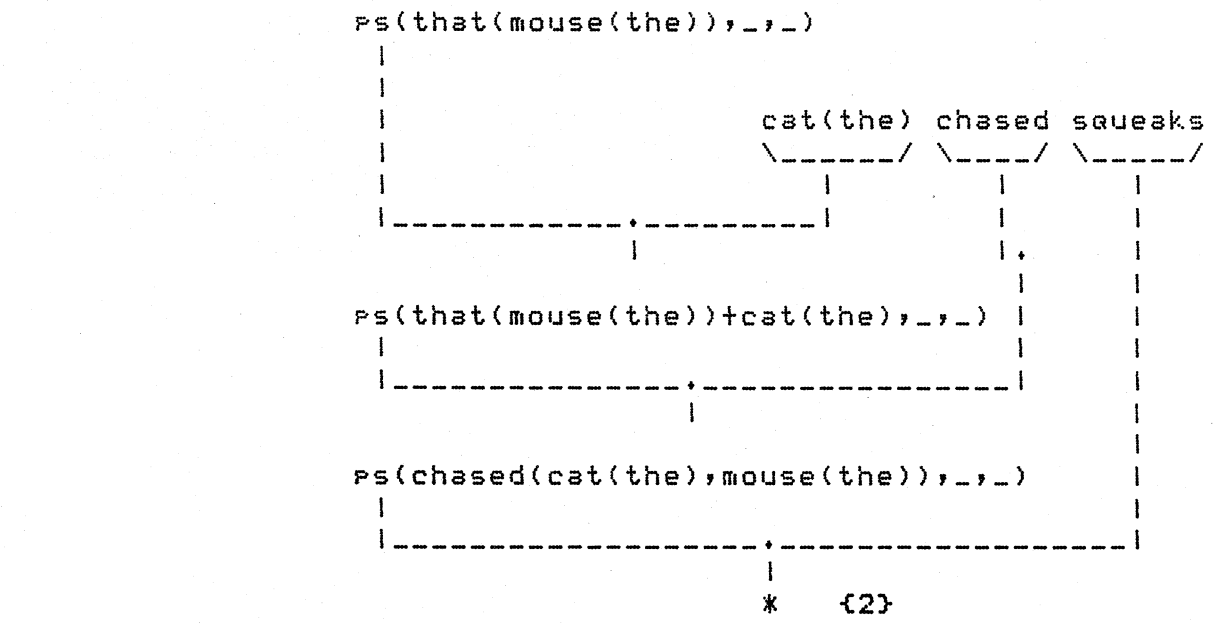
The second example shows that a sentence violating the Ross complex-NP constraint will not be accepted by **SPIRAL** - for ease of exposition the determiners are dropped.

Input sentence : the mouse that the cat chased squeaks

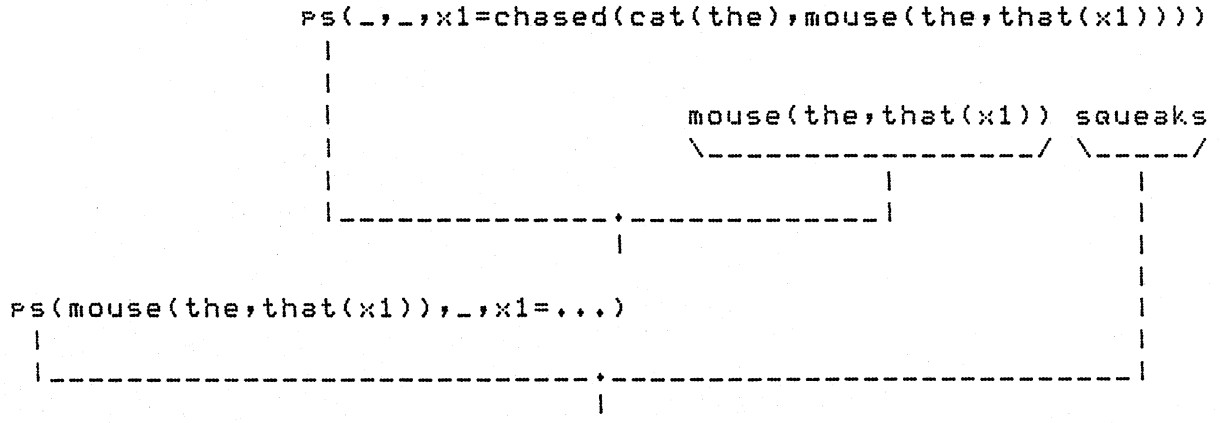
After 1st level : mouse(the) that cat(the) chased squeaks

2nd, 3rd levels : _____/ {1}

recursive analysis on :



end of recursive analysis {3}



ps(squeaks(mouse(the,that(x1))),_,
x1=chased(cat(the),mouse(the,that(x1))))

Result : squeaks(mouse(the,that(x1))) &
x1=chased(cat(the),mouse(the,that(x1)))

{1} - the relative will be analysed throush a recursive analysis.

{2} - 'squeaks' cannot be added to 'chased(the cat,the mouse)' because a phrase cannot have two main verbs, resulting in a failure at the 3rd level and the end of the recursive analysis.

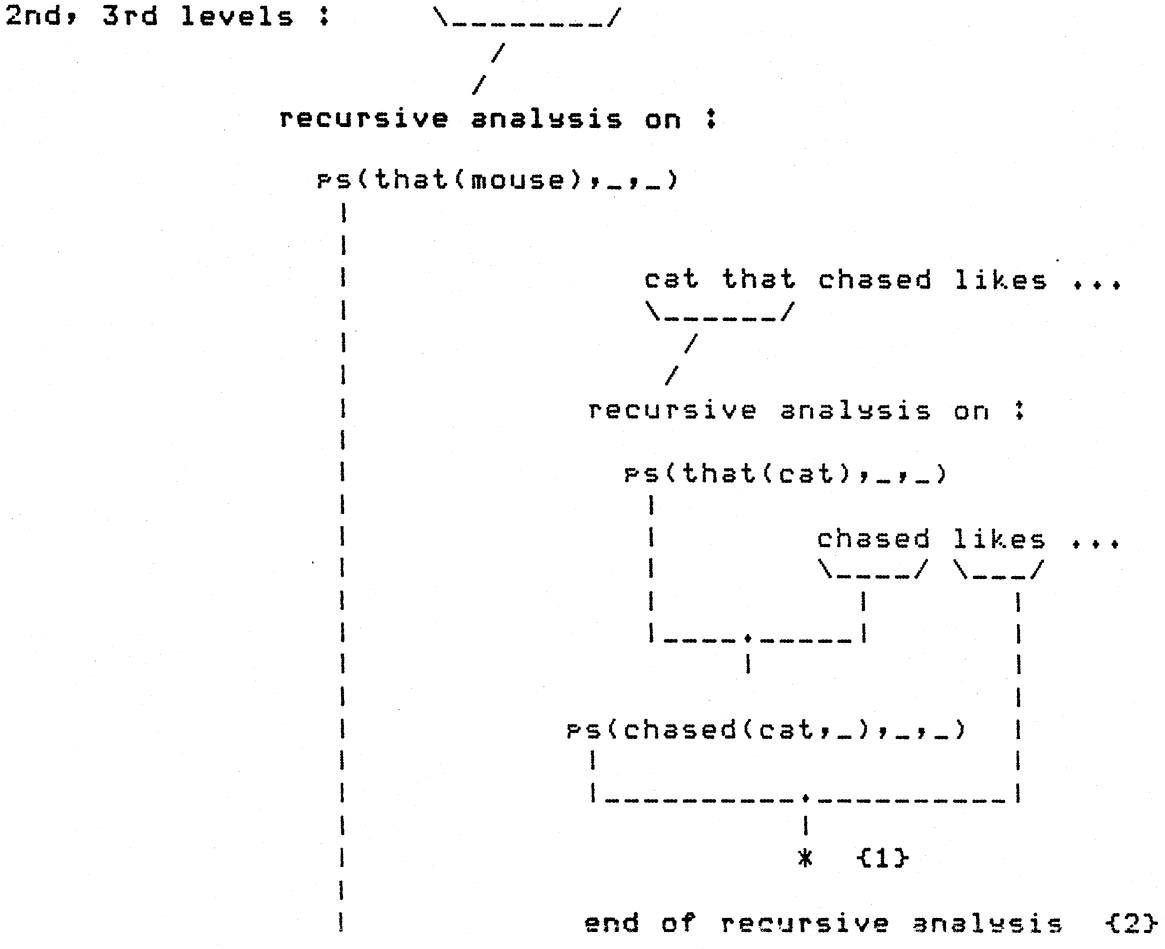
{3} - from the recursive analysis results a phrase structure that is passed as a subphrase to the 3rd level by the rule launching the recursive analysis ; this rule is also responsible for the binding of 'that(x1)' to the noun 'mouse' and for the anteposition of this noun to the remnant sentence.

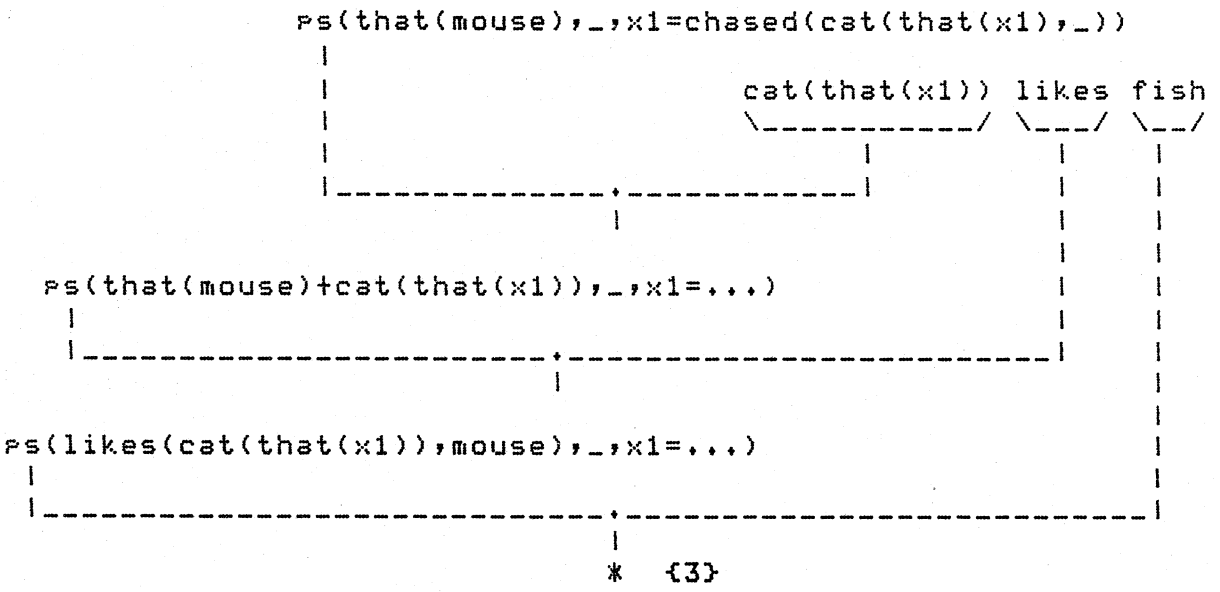
Input sentence :

the mouse that the cat that chased likes fish squeaks

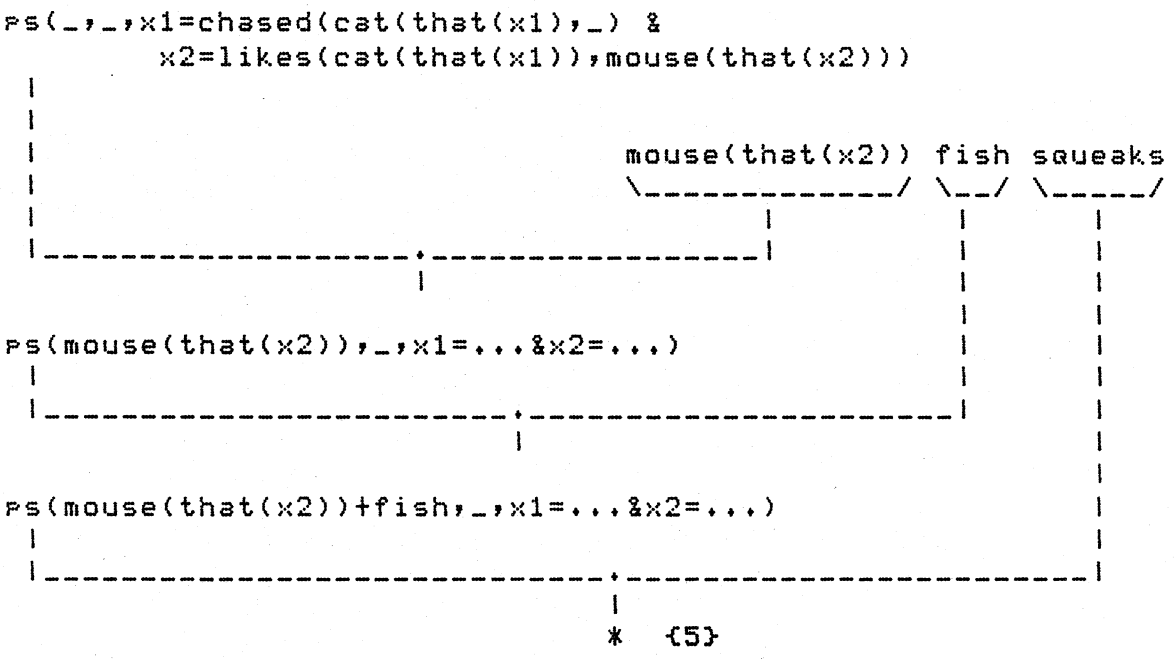
After 1st level : mouse that cat that chased likes fish squeaks

2nd, 3rd levels :





end of recursive analysis {4}



{1} - a phrase cannot have two main verbs, then 'likes' can not be added to 'chased(cat, something)'.

{2} - the subphrase just found is added to the phrase structure that already existed. Note that 'chased' is treated as transitive though with a direct object not stated - a common situation with certain verbs.

{3} - 'fish' cannot be added to 'likes(cat,mouse)' by the

reason in (1) above.

(4) - the two subphrases found so far are conjoined.

(5) - the analysis fails as 'squeaks' is intransitive.

If 'fishes' occurred instead of 'fish' and if a mouse could in any way fish squeaks (and in poetry - at least - this is obviously possible), the following analysis would be arrived at :

```
fishes(the mouse (that (
                    likes(the cat(that chased something),
                          the mouse)),
            squeaks)
```

To illustrate other capabilities of the syntactic analyser in SPIRAL some sentences that it accepts are listed below, the last of which because a direct translation from Portuguese is not correct in English - words within parentheses do not appear in the Portuguese version.

the author wrote a book in 1910.

in 1875 the author decided to write a book.

the works that the author wrote are for the piano.

the author that wrote in Venice a book.

the work that in 1920 was written by the author.

the author that was born in London and whose work was written in Paris.

the author whose work was written in the 18th century.

the piano is the instrument for which the work was written.

the authors in whose centuries works have been written.

the work A is older than the work B.

who wrote books ?

who wrote the oldest book ?

which are the works that were written in the 20th century ?

which are the works from the 19th century ?

in which century was born the author ?

the author wrote all his works in London.

the author that was born in the place where (he) wrote his works.

Lexical Analysis

In order to use dictionaries similar in content to current dictionary books (and this should be a goal for any natural language interface) some kind of suffix analysis must be performed at the lexical recognition stage. This need is still more urgent when analysing languages like Portuguese or French that make systematic use of inflections and conjugations, for substantial savings in dictionary space can then be gleaned.

To this end, I built (together with Antonio Porto, and much in the vein of [P. Sabatier, J.F. Pique 82]) a lexical analyser using a set of inflection/conjugation rules along with a dictionary containing word roots, words that constitute exceptions to the given set of rules or that are not described by them, and words that have no suffixes. For each input word (represented by the list of its characters in reverse order) the analyser tries a direct dictionary entry and subsequently (either by a failure in this attempt, or by a failure at the syntactic or semantic levels) performs suffix analysis. The current set of rules for Portuguese (some 80 Prolog clauses) covers 4 verbal conjugations in the 1st and 3rd persons, singular and plural, 4 tenses and pronominal conjugation for all this, as well as almost all inflections according to gender and number - some 17 different forms of plural. Typically a clause specifying a verb root implicitly defines some 68 different forms for it !

The counterparts to the dictionary compactness attained by this method are :

- some problems of representation duplication if word surface representation is to be kept for future use

- the dilemma of either allowing strange words to be accepted as valid by the inflection/conjugation rules, or burdenning the lexical analyser with tests

- an unfelt loss of efficiency

Concerning the dilemma above, if one accepts that the user should be responsible for the use of, e.g., 'writed' instead of 'wrote', there should be no damage if the natural language interface understands it according to the general rules. This is all the more so if the natural language interface provides a paraphrase of what has been understood after analysing a sentence - a research direction that will be taken soon. Obviously, for those not sharing this point of view there remains the possibility of providing tests to filter erroneous words.

Lexical ambiguity is treated by backtracking from the syntactic analyser. Some experiments on co-routining the lexical and syntactic analysers were made with some success by Antonio Porto using his ideas on control [A. Porto 82], and will be pursued in due course.

Semantic Analysis

For sake of modularity and generality, the semantic analyser uses an intermediate semantic representation (ISR) form to build a Prolog goal expression from a syntactic structure. An ISR form consists of Prolog goals, object (in general, entity) descriptions and auxiliar pseudo-goals (used to pass information while building the ISR form). Object descriptions are used the same way as in [A. Walker, A. Porto 83] ; in SPIRAL they occur under the form of a 3-place functor

o(Type:Var, Quant, Cond)

containing the object type, the Prolog variable associated with it, its quantification, and a defining condition in ISR that may contain other object definitions.

For instance, to the sentence

'the works from the authors of each century'

corresponds the following ISR expression and Prolog goal

```
o(work:W, each,
        o(author:A, each,
                o(century:C, each, _) &
                author(A,D) & century(D,C) ) &
        work(W,A) )
```

```
set(work/author/century) : Swac <-
        all(Swa/C,
            gen_cent(C) &
                all(Sw/A,
                    author(A,D) & century(D,C) &
                        all(W, work(W,A), Sw),
                    Swa),
            Swac)
```

where 'all' is the predicate defined in [L. Moniz Pereira, A. Porto 81] and 'gen_cent' is a generator of suitable century values.

ISR expressions are built from the syntactic structure by some general predicates, plus a separate set of application dependent ones, that define the semantics for verb and its complements, verb and its arguments, and noun and its complements. Writing such predicates for a particular application is made easy by the use of pseudo-goals and some pre-defined predicates coping with them (adding a Prolog goal to a condition, substituting a pseudo-goal by a Prolog goal, choosing and inserting Prolog goals from a list, and so forth).

When translating an ISR expression to a Prolog one, scoping problems concerning **distributive quantifiers** (such as 'each') and **aggregations** (such as 'average') [F. Pereira 83] are dealt with.

Efficiency and Future Work

SPIRAL has been implemented using the RT-11 Prolog interpreter by Clocksin, Mellish, Byrd and Fisher [W. Clocksin, C. Mellish, R. Fisher 80] and adapted by A. Porto and I to run under an **RT-11 Extended Memory** environment on a **PDP-11/23** machine with floppy-disks. The program currently occupies some **15K** (16-bit) words (in terms of nicely presented Prolog text about **23** pages as follows : 5 for the lexical analyser (including a common dictionary), 9 for the syntactic analyser, 3 for the semantic one, and 6 for the current application dependent parts - the application dictionary included). The remaining **5K** left free are what is needed as workspace. Future extensions under these conditions may force the use of a two-job partition as in [L. Moniz Pereira, P. Sabatier, E. Oliveira 82] or [L.M. Pereira, A. Porto 82] - it is no novelty that a **PDP-11/23** is a somewhat restricted machine !

Response times, though no exact benchmarks have been made, are comparable to those described in [L.M. Pereira, P. Sabatier, E. Oliveira 82] or [L.M. Pereira, A. Porto 82] and vary from less than 1 second for most sentences, to 10 or more seconds for very complex ones - these times are better than those obtained by a Lisp program that attempts to understand noun compounds, running on a PDP 2060 (it takes some 5 seconds to analyse 'glass wine glass') [D.B. McDonald 82].

These results are quite satisfactory taking into account the machine used - whenever the 5th generation machines [T. Motooka (ed.) 82], [D. Warren 82] become a reality this section will stand as an example of concern with anachronistic values.

As already stated, **SPIRAL** is thought of as an open (as any spiral!) kernel for a natural language interface and this means that many research directions are open to further extend its abilities. Among them, those concerned with the following, to be explored soon :

- actions to be performed when a sentence cannot be understood or is ambiguous (dialogues with the user and paraphrasing will be sought)

- means to help configurate the interface to a new domain (wherever possible those used in [M. Filgueiras, L. Moniz Pereira 82])

Acknowledgements

Antonio Porto, with whom I had many fruitful discussions on these (and other) matters, gave me an invaluable help on the development of the semantic analyser.

Luis Moniz Pereira is thanked for all his enlightening comments.

References

A. Colmerauer

75 Les Grammaires de Metamorphose
Groupe d'Intelligence Artificielle
Universite' d'Aix-Marseille II

78 Metamorphosis Grammars
in Natural Language Communication with Computers
ed. L. Bolc
Lecture Notes in Computer Science
Springer Verlag

W. Clocksin, C. Mellish, R. Fisher

80 The RT-11 Prolog System (Version NU7 with Top Level)
Software Report 5a (2nd ed.)
Department of Artificial Intelligence
University of Edinburgh

V. Dahl

77 Un Systeme Deductif d'Interrogation de Banque de Donnees en Espagnol
These de 3eme Cycle
Groupe d'Intelligence Artificielle
Universite' d'Aix-Marseille II

M. Filgueiras, L. Moniz Pereira

82 Relational Data Bases `a la carte
Departamento de Informatica, Universidade Nova de Lisboa

Proceedings of the Logic Programming Workshop 83, Portugal

G. Hirst

81 Anaphora in Natural Language Understanding : a survey
Lecture Notes in Computer Science 119
Springer Verlag

M. Kay

80 Algorithm Schemata and Data Structures in Syntactic Processing
Palo Alto Research Center, Xerox

M. McCord

80 Using Slots and Modifiers in Logic Grammars for Natural Language
Technical Report No. 69A-80
Computer Science Department, University of Kentucky

in Artificial Intelligence 18(3), 1982

81 Focalizers, the Scoping Problem and Semantic Interpretation Rules in Logic Grammars
Technical Report No. 81-81
Computer Science Department, University of Kentucky

in Proceedings of the Workshop on Logic Programming for Intelligent Systems
Logicon Inc., 1981

D. B. McDonald

82 Understanding Noun Compounds
Ph.D. Thesis
Department of Computer Science, Carnegie-Mellon University

L. Moniz Pereira, A. Porto

81 All Solutions
Logic Programming Newsletter, 2, Autumn 81

82 A Prolog Implementation of a Large System on a Small Machine
Proceedings of the First International Logic Programming Conference, Marseille, France

L. Moniz Pereira, P. Sabatier, E. Oliveira

82 ORBI : An Expert-System for Environmental Resource Evaluation through Natural Language
Proceedings of the First International Logic Programming Conference, Marseille, France

T. Motooka (ed.)

82 Proceedings of the International Conference on the Fifth Generation Computer Systems, Tokyo, Japan, 1981
North-Holland

R. Pasero

73 Representation du Francais en Logique du Premier Ordre en Vue de Dialoguer avec un Ordinateur
These de 3eme Cycle, Groupe d'Intelligence Artificielle
Universite' d'Aix-Marseille II

F. Pereira

81 Extraposition Grammars
American Journal of Computational Linguistics, vol. 7, 4

83 Logic for Natural Language Analysis
Technical Note 275
SRI International

F. Pereira, D. Warren

80 Definite Clause Grammars for Language Analysis -
A Survey of the Formalism and a Comparison with
Augmented Transition Networks
Artificial Intelligence, 13

J. F. Pique, P. Sabatier

82 An Informative, Adaptable and Efficient Natural
Language Consultable Data Base System
Proceedings of the European Conference on Artificial
Intelligence, Orsay, France

A. Porto

82 Epilog : A Language for Extended Programming in Logic
Proceedings of the First International Logic Programming
Conference, Marseille, France

A. Walker, A. Porto

83 KB01 : A Knowledge Based Garden Store Assistant
Proceedings of the Logic Programming Workshop 83, Portugal

to appear as an IBM Research Report

D. Warren

82 A View of the Fifth Generation and Its Impact
Technical Note 265
SRI International

AN OPERATIONAL ALGEBRAIC SEMANTICS OF PROLOG PROGRAMS

DERANSART Pierre

INRIA

Domaine de Voluceau - Rocquencourt

BP 105

78153 Le Chesnay

April 1983

Abstract : We shall show that the resolution strategy implemented in most of the PROLOG interpretors may be equivalently viewed as a particular equation solving in an associated algebraic specification. We suggest and illustrate possible applications of this approach to analysis of PROLOG programs.

Keywords : PROLOG - ALGEBRAIC ABSTRACT DATA TYPES - PROGRAMMING ENVIRONMENT.

PRESENTATION (Long abstract)

We shall show that the resolution strategy implemented in most of the PROLOG interpretors may be equivalently viewed as a particular equation solving in an associated algebraic specification. We suggest and illustrate possible applications of this approach to analysis of PROLOG programs.

The basic point of this work is a rigorous correspondence between a PROLOG program and his translation -if any- into an algebraic specification. Most of the studies about PROLOG semantics [Vako76] are devoted to the "pure PROLOG", i.e. PROLOG (restricted to first order logic programming) without "control" nor evaluable predicates. By "control" we mean two things : the famous "cut" operator and the strategy of choosing the clauses and the litterals to be solved. Our aim is to integrate the second element into the semantics in order to get a kind of operational semantics taking in account this aspect of the control.

In fact, the logical part of a PROLOG program get rise, in numerous programs, to a rapid understanding and easy verification of the program properties, analogous to partial correctness proof of programs [ClTa77]. But halting problems or invertibility aspects give unexpected and sometimes difficult to manage behaviours, even of simple programs. A programmer is not only interested to know if his goal is a logical consequence of the axioms, but essentially interested to know how his goal will be satisfied, if there is no infinitely nested loop or if he will obtain all the solutions (the completness in this sense has to be defined), in which order, etc... . Lot of these questions have an empirical answer, without any aid of known semantic models.

On the other side, algebraic specifications have been extensively studied with practical (operational) and semantical points of view [ADJ78, GH78]. Some specifications can be viewed as equational theories. In our approach, specifications are only viewed as a practical way to describe environments and programs in the same formalism and are limited to so-called "specification with constructors" similar to equational theories with constructors of [HH80] but with conditional axioms.

This work should have various applications. Behaviour studies of PROLOG programs or equivalent program transformations are part of them. Some examples of non trivial programs have been studied by this method, like permutations, eight-queens problem and Baxter example [Ba81]. Practical limitations of this approach come from the type of conditional axioms which can be easily studied.

Each time the specification is a canonical and complete TRS the situation is quite agreable : it is in fact possible to use directly properties of the specification in order to transform or modify the programs.

In the general case of equalitarian axioms, the main difficulties seem to come out from the few existing works on such axiomatisation and the equalitarian TRS that can be defined on. Some constraints can be given such that numerous interesting programs fall down in this class, but the practical study of derivations remains difficult. It seems to us that a usefull tool could be a PROLOG programming environment in which narrowing of transformed goals could be formally analyzed. Nevertheless, difficulties come from two levels :

1) Semantical level : in all the cases, the obtained specification is a partial algebra, because of the manipulation of partial functions.

2) Operational level : the generalized TRS did not have been enough studied until now [Re82, Ka83]. The corresponding notions of canonical and complete TRS remain to be better known.

It seems to us that these difficulties reflect well the situation we feel in PROLOG programming : difficulties to specify the error cases in a satisfactory manner (frequently only positive cases are specified), quasi-impossibilities to have a clear idea of the set of produced solutions, his completness, except by personal conviction of the programmer.

Finally, our study can be viewed from a dual point of view :

- Conversion of an abstract data type into a PROLOG program. So it is a way to get a direct and efficient implementation of the transitive

Various papers are dealing with correspondence between specifications or functional programming and PROLOG [VaMa81, BD81]. Generally the correspondence shows that PROLOG is a suitable specification approach. But the correspondence is not always very precisely stated.

We will define a strict correspondence by the following manner :

- To any predicate we associate a functional decomposition. A predicate of arity n is said 1-decomposable, iff there exists an equivalent function of arity n-1 with corresponding domains. This notion can be generalized into k-decomposability.

- To any PROLOG program that can have a functional decomposition, it is possible to associate a specification with constructors. If there is no functional decomposition, the transformation is trivial and of few interest. In all the cases the transformation is a one to one correspondence.

- We show that the resolution of a PROLOG goal, using the usual interpretor strategy, is exactly the same as to solve an equation (the transformed goal) using a strategy called ℓ-i-resolution. If the specification is an equational theory, this problem reduces to an unification problem solved by ℓ-i-resolution (this approach uses a relation called "narrowing").

- Finally we use this transformation in order to study the solutions of the goal equations, in particular the capacity of invertibility of a program.

This approach gives an operational characterization of PROLOG programs admitting such an analysis (functional decomposition plus specification with particular properties). The approach is completely symetric and the obtained class is not restrictive : it has the power of computable functions. So it is possible to have dual point of view : in one sense PROLOG realizes an operational implementation of conditional algebraic specifications, on the other the models of the specification can be models of the PROLOG program.

closure of the ℓ-i-narrowing. In this case we shall speak of "compilation of specifications into a PROLOG program".

- Conversion of a PROLOG program into an abstract data type. This is a way to verify the original program structure (by typing the elements, verifying completness...) and, eventually, to modify it using correct transformations.

BIBLIOGRAPHY

[ADJ78] GOGUEN J.A., THATCHER J.W., WAGNER E.G.
An Initial Algebra Approach to the Specification Correctness and Implementation of Abstract Data Types in "Current Trends in Programming Methodology".
Chap. IV (R. Yeh, ed) - pp. 80-149 - Prenctice Ha-1 1978.

[Ba81] BAXTER L.
The Versatility of PROLOG.
SIGPLAN Notices - York University.

[BD81] BERGMAN M., DERANSART P.
Abstract Data Type and Rewriting System : Application to the Programming of Algebraic Abstract Data Types in PROLOG.
CAAP 81 - Trees in Algebra and Programming - 6th Colloquium - March 81 - LNCS 112.

[ClTa77] CLARK K., TARNLUND S.A.
A First Order Theory of Data and Programs.
Proc. IFIP 77 - pp. 939-944.

[GH78] GUTTAG J.V., HORNING J.J.
The Algebraic Specification of Abstract Data Types.
Acta Informatica 10 (1978) - pp. 27-52.

[HH80] HUET G., HULLOT J.M.
Proofs by Induction in Equational Theories with constructors.
Rapport INRIA n° 28.

[HO80] HUET G., OPPEN D.
Equations and Rewrite Rules : a Survey.
"Formal Languages : Perspectives and Open Problems".
Ed. Book R. - Academic Press 1980 - Also TR-CSL-111.
SRI International - January 1980.

[Ka83] KAPLAN S.
An Abstract Data Type Specification Language.
(French) Thesis - University of Orsay - February 3, 1983.

[Re82] REMY J.L.
Conditional Rewrite Rules Systems and Applications to Algebraic Data Types.
(French) Doctorat Thesis - University of Nancy - July 13, 1982.

[VaKo76] VAN EMDEN M.H., KOWALSKI R.
The Semantics of Predicate Logic as Programming Language.
JACM 23 - 1976 - pp. 733-742.

[VaMa81] VAN EMDEN M.H., MAIBAUM T.S.E.
Equations compared with Clauses for Specification of Abstract Data Types.
Advance in Data Base Theory - V1 - Ed. Gallaire, Minker, Nicolas - Plenum Press. - 1981.

443

INTERNATIONAL COMPUTERS LIMITED
SYSTEMS STRATEGY CENTRE

REF: EB 83/6
DATE: 4th May 83

FINITE COMPUTATION PRINCIPLE
An Alternative Method Of Adapting Resolution For Logic Programming

E. Babb

ABSTRACT

Currently PROLOG implements resolution by means of symbolic substitution. The result is that **symbolic** operations (eg on lists) in PROLOG are reversible, whilst **data** operations (eg arithmetic) are not. This paper proposes an adaption to the resolution principle called the **Finite Computation Principle (FCP).** Using FCP, symbolic substitution is still available but is performed by a special predicate.

FCP gives the two important benefits of **Order independence** and **control over infinite processes.** In addition, FCP improves reversibility and simplifies the connection of logic to existing languages.

A logic language called Prolog M has been implemented using FCP. This provides standard negation, disjunction, conjunction, universal quantifiers and existential quantifiers. An important feature of the implementation is that if two Prolog M programs are **equivalent** according to the tautologies of Predicate Calculus, then these two programs will generate **identical** answers.

1. INTRODUCTION

During the 1970s the author was actively involved in the hardware and software design of the ICL Content Addressable File Store (CAFS). It was during this period that a method of making queries to a database without the reference to relation or file names [2] was proposed. This shorthand was made possible by including a limited mathematical model in the language interpreter. This model being made up of joins of relations. This technique has proved successful with database users but was limited to joins of physical relations - i.e. conjunctions of predicates. It was as a result of trying to generalise this model that it was realised how useful Prolog might be in this area.

Prolog is order sensitive. Despite the name, Prolog is not a true logic language and the database query below must be written in a particular order.

Manweight(x,w) , w < 20	**works!**
w < 20 , Manweight(x,w)	**Errors!**

In a database query, it is essential that the terms can be written in any order. Warren [7] recognises this in his CHAT80 database system. In CHAT80 additional features are included to allow order independence.

Prolog is very likely to go infinite. For example, Define **Append** in the standard way and then make the following queries:

Append(x,(4),w)	**prints infinite formula or infinite number of values**
Append((4),y,w)	**prints w = (4.y) or infinite list of values.**
Append((4),y,w) , y=w	**just does nothing!**

In this last example, Prolog is trapped in a silent contradiction. One predicate generating an infinite number of instances of the y and w variables, whilst the subsequent predicate **y = w** always fails. In a large machine with almost unlimited storage resources such an infinite contradiction could be a very expensive bug.

The Finite Computation Principle (FCP) makes two things possible:

True order independence
No infinite processes

Ordinary resolution is still available using a pattern matching predicate. However, FCP allows operations on lists or sets of data to be carried out more securely.

The power of FCP appears when it is used in recursive definitions. Thus, most of the paper is concerned with explaining the operation of a number of key examples - in particular **APPEND**. The paper then hints at what may be possible in the future.

1.1 Notation

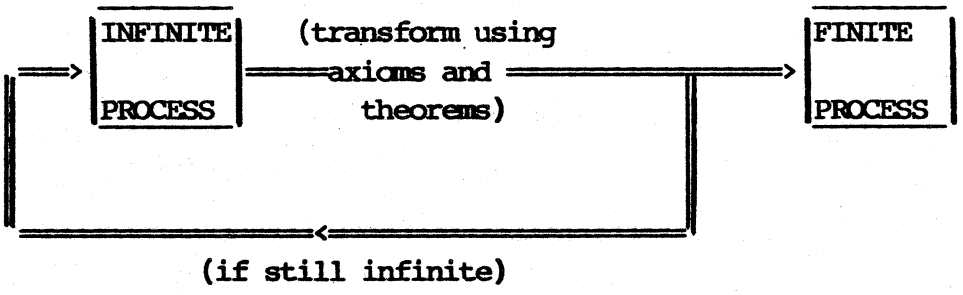
Prolog M uses a LISP like notation for predicates. However, for clarity this paper uses the conventional mathematical notation. Nevertheless, so that the flavour of Prolog M is not lost, the Prolog M syntax is often written along side in curly brackets.

Prolog M means Prolog with a Model[1,2]. It is hoped to describe the model aspect of Prolog M in a forthcoming issue of the ICL technical journal.

2. THE FINITE COMPUTATION PRINCIPLE (FCP)

The basic notion of the Finite Computation Principle is one that arises from the nature of logic programming. Some expressions written in a logic programming language may result in the infinite generation of data or some other endless process. FCP seeks to flag up infinite processes and put off their evaluation until the last possible moment by which time they may become finite processes as a result of information returned from other processes. **This is done by including in every built in predicate a test for the condition that makes it infinite.**

FCP detects that a process is infinite and then applies axioms and theorems to eventually create a finite process in the manner indicated below:



A process is either an atomic predicate, meta predicate (such as conjunction) or a user defined predicate. Current Prolog M only uses the axioms of logic to attempt to render a process finite.

To a limited degree CHAT80 uses something similar to FCP to delay the execution of negated predicates. The crucial feature about FCP is that every predicate should be able to identify the conditions that might make it infinite. It is then possible, as indicated above, to delay the execution of infinite predicates, even in recursive definitions.

In a fully working version of Prolog M, the optimal delaying of processes would also be included, as indeed it was in the ICL CAFS database system [3].

447

3. ATOMIC PREDICATES

Atomic predicates flag **infinite** if their use would generate an infinite solution set. For example:

x = 6	**FINITE:**	**only one solution**
x = y	**INFINITE:**	**(1,1)(2,2)(3,3)...**
2 = 4	**FINITE:**	**no solutions**
x = y + z	**INFINITE:**	**(1=1+0)(2=1+1)....**
(2,3) = u . x	**FINITE:**	**u is head of list ie 2 and x is tail of list ie (3)**

In Prolog M these have the syntax (=,x,6), (=,x,y), (=,2,4), (+,x,y,z), (.,(2,3),u,x) respectively.

Infinity is flagged by including in the definition of the predicate, code which will set an infinite flag for certain combinations of free variables.

4. LEFT TO RIGHT PREDICATE

Predicates are normally executed from left to right:

w < 20 , Manweight(x,w) { ((<,w,20)(Manweight,x,w)) }

Provided both are finite then the whole expression is finite. In this case the first atomic predicate is infinite and so the whole expression is infinite.

448

5. CONJUNCTION

Conjunction allows the machine to apply the axioms of logic to determine a finite ordering. Thus, if we write the following:

w < 20 & Manweight(x,w) { (&(<,w,20)(Manweight,x,w)) }

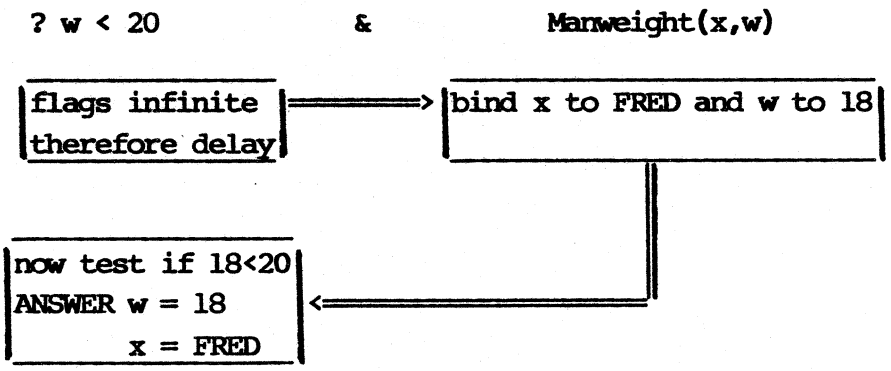
The machine will attempt execution from left to right:

w < 20 , Manweight(x,w)

The result is infinite and so using the axiom **A & B <-> B & A** the reverse ordering is tried:

Manweight(x,w) , w < 20

If we assume manweight has instance FRED,18 then execution is as follows:



If there had been nested conjunctions, these would be collapsed down so that **((A & B) & C)** would be replaced by **(A & B & C)**.

6. DISJUNCTION

Disjunctions of two or more predicates are executed as two quite separate processes. Thus:

```
(x = 3 or x = 4) & f(x)          { (&(or(=,x,3)(=,x,4))(f,x)) }
```

is executed as two processes:

```
x = 3 & f(x)                     { (&(=,x,3)(f,x)) }
x = 4 & f(x)                     { (&(=,x,4)(f,x)) }
```

First, x is given the value 3 and f is executed. Second x is given the value 4 and f is again executed. For a disjunction to be finite both these processes must be finite.

7. EXISTENTIAL QUANTIFICATION

If there exist values of x1,x2,... that satisfy an expression p then the expression q is executed:

```
some(x1,x2,...)(p) , q           { (some (x1,x2,...)p) }
```

We evaluate this by forcing x1,x2,.. to be variables local to p. Thus, they start off initially as free variables. If there are instances of these variables locally in p then the expression q is executed.

8. NEGATION

Negation is implemented by transforming the negated expression so that the **not** is moved to a subexpression using one of the three axioms:

not(p&q) —> notp or notq	{ (not(&,p,q)) --> (or(not,p)(not,q)) }
not(p or q) —> notp & notq	{ (not(or,p,q))--> (&,(not,p)(not,q)) }
notnotp —> p	{ (not(not,p))-->p }

Eventually, the expression cannot be changed because none of these transforms can be applied. It will then be found that p is either an atomic predicate or an existential quantifier. We therefore actually execute the **not(p)** predicate. The **not** means no instances. Thus, the **not** is executed by checking that the predicate p has indeed no instances. If this is the case, then we allow execution of any statements that follow. This is "Negation as Failure" [4]. In the example:

not6=7 & x = 9 { (&(not(=,6,7))(=,x,9)) }

six does not equal seven, there is no instance, and so the next term x=9 is executed.

Negation has its own special infinities. A negation is finite only if all the free variables of p are externally bound. Thus, the free variable of the expression **x = 6** is x. Therefore, for **not x = 6** to be finite, x must be bound at the time when the **not** is executed. Clearly, if x had been free then there would be an infinite number of x values not equal to six. Again by trapping this infinite case it is possible for other processes to bind x.

8.1 SPECIAL CASE

Suppose an expression **not p** has free variables and is therefore infinite. It is sometimes possible, when the expression p starts with the quantifier **some**, to manipulate **p** to give a new expression **p'** which generates the bindings for these free variables. The resulting expression **p'&not p** now being finite. For example, the infinite statement

```
not some(x)(r(x)&not h(x,y))
                    {(not(some(x)(&,(r,x)(not(h,x,y))))) }
```

can be rendered finite by moving **h(x,y)**, the negated term in p, outside:

```
some(x)h(x,y)  &  not some(x)(r(x)&not h(x,y))
```

This new term now creates a finite set of bindings for y. A general theorem for transforming p to p' is given in reference [1].

9. UNIVERSAL QUANTIFIERS

Universal quantifiers are equivalent to negative existential quantifiers and so they are transformed before execution using the axiom:

```
all (x1,x2,...)(p) --> notsome(x1,x2,...)(not p)

                    { (all(x1,...)p)-->(not(some(x1,..)(not p))) }
```

452

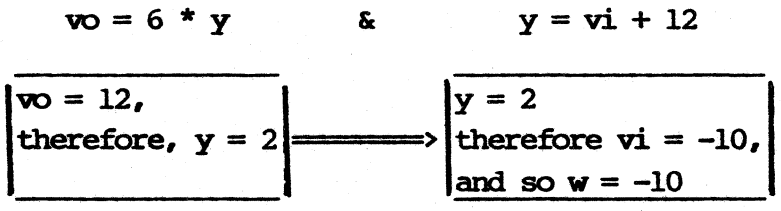
10. DEFINITIONS

Definitions allow complex expressions to be represented by a single predicate. Consider the definition:

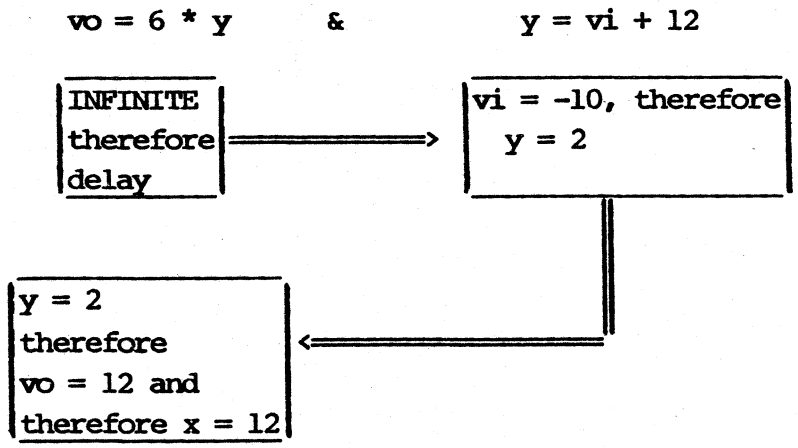
amplifier(vo,vi) <- vo = 6 * y & y = vi + 12

When this is called using the query ?amplifier(12,w), the variable vo inside the definition takes on the value 12 while the variable vi points to an identical location in store to w and hence become equivalent. The variable y is local to the definition and so there is an implicite existential quantifier.

This definition is fully reversible, so we can either ask the question **?amplifier(12,w):**



or the reverse question **?amplifier(x,-10):**



We can now define another predicate representing two amplifiers in cascade and still have reversibility:

amps(vo,vi) <- amplifier(vo,x) & amplifier(x,vi)

Term 1 or 2 being automatically selected depending whether vo or vi is bound.

10.1 RECURSIVE DEFINITIONS

Consider the operation of **append**. This can be defined by the single recursive definition which appends list x to list y to give list z:

```
append(x,y,z) <-    x = ()  &  y = z   or
                    x = u.x' & z=u.z' & append(x',y,z')
```

This definition states that if x is an empty list then lists y and z are equal. Otherwise, if we strip u off lists x and z then the remaining lists x' and z' are related by the append predicate. When used in recursion neither **or** nor **&** are order independent. This is because recursive calls to the append predicate always have the possibility of being infinite and so should always be written last. It may be sensible in some future implementation to automatically place such recursive calls last thus restoring order independence.

Using FCP this definition gives the following results. Readers interested in the details of the execution are refered to the appropriate appendix.

?append((2),(3),z)	**z = (2,3)**	**APPENDIX 1**
?append(x,y,(2,3))	**x = () y = (2,3)**	
	x = (2) y = (3)	**APPENDIX 2**
	x = (2,3) y = ()	
?append(x,(2),z)	**infinite flag set**	**APPENDIX 3**
?append((2),y,z)	**infinite flag set**	

Notice how FCP correctly traps the infinite process. Contradictions as mentioned earlier can therefore be trapped before execution:

```
?append((2),y,z) & y = z
    flags infinite
```

Without this facility a naive user could be faced with some expensive computer bills!

10.2 EXPLICITE TRAPPING OF INFINITE PROCESSES

Suppose we define a factorial predicate **fact'(x,n)** which gives the factorial x of a number n. When x and n are both free we find that **fact'** executes an **actual** infinite loop. To prevent this infinite loop we precede **fact'** by the predicate **free:**

fact(x,n) <- free(x,n) , fact'(x,n)

The predicate **free** flags infinite if all its arguments are free. Thus by detecting that x and n are both free, factorial is now secure against infinite loops.

Notice that **append** did not require any such trap to stay finite.

11. THE FUTURE

Prolog M uses the axioms of logic to transform an infinite expression to a finite expression. However, the capabilities of the language could be considerably extended if the user were also able to define his own infinite to finite axioms and theorems. Below is a simple example of an axiom to allow a natural way of writing a range of numbers:

x > xmin & x < xmax & integer(x) <- range(x,xmin,xmax)

{ (define (&(>,x,xmin)(<,x,xmax)(integer,x)) (range,x,xmin,xmax))}

It is now possible to write the query:

?x > 1 & x < 12 & integer(x)

and obtain the integers 2,3,...10,11 using the range predicate.

We can define new functions in the same way that we can bind variables to values. For example:

quadfn(x) = "x*x + 2*x - 4"

{(=,quadfn,'(lambda(x)(plus(times,x,x)(times,2,x),-4)))}

binds the function variable **quadfn** to **"x*x + 2*x - 4"** using a lambda expression. The function quadfn can then be used in an equality predicate:

y = quadfn(x) { (=,y,(quadfn,x)) }

Unlike normal equality, this is finite only if **x** is bound.

Predicates are often defined in terms of a forward and reverse function. A reversible quadratic function **quad(y,x)** is defined as the conjunction of **quadfn** and **quadreversefn.** The appropriate function being chosen by FCP.

12. CONCLUSION

Prolog M is still in its infancy. There are at least three important questions left unanswered:

1. By trapping the generation of infinite formulas, will FCP make conventional resolution more flexible?

2. Can trace facilities easily explain why programs are infinite?

3. Can we easily include user defined theorems?

ACKNOWLEDGEMENTS

The author is grateful to the late Roy Mitchell, Vic Maller, Norman Truman, Martin Stears and the other members of the ICL Systems Strategy Centre, Stevenage who have helped to formulate and develop the ideas in this paper.

REFERENCES

[1] BABB E — SYSTEM MODELLING LANGUAGE (SML) For Enquiries to a Business or Scientific Model. ICL Technical Note TN 82/1, 1982

[2] BABB E — Joined Normal Form: A storage encoding for relational databases. ACM Trans. on Database Systems. December 1982

[3] BABB E — Implementing a Relational Database by means of Specialised Hardware. ACM TODS, June 1979

[4] CLARK K & TARNLUND S.A.(Eds) — Logic Programming, Academic Press, 1982

[5] CLOCKSIN W & MELLISH C — Programming in PROLOG, Springer-Verlag, 1981

[6] KOWALSKI R A — Logic For Problem Solving, North Holland 1979

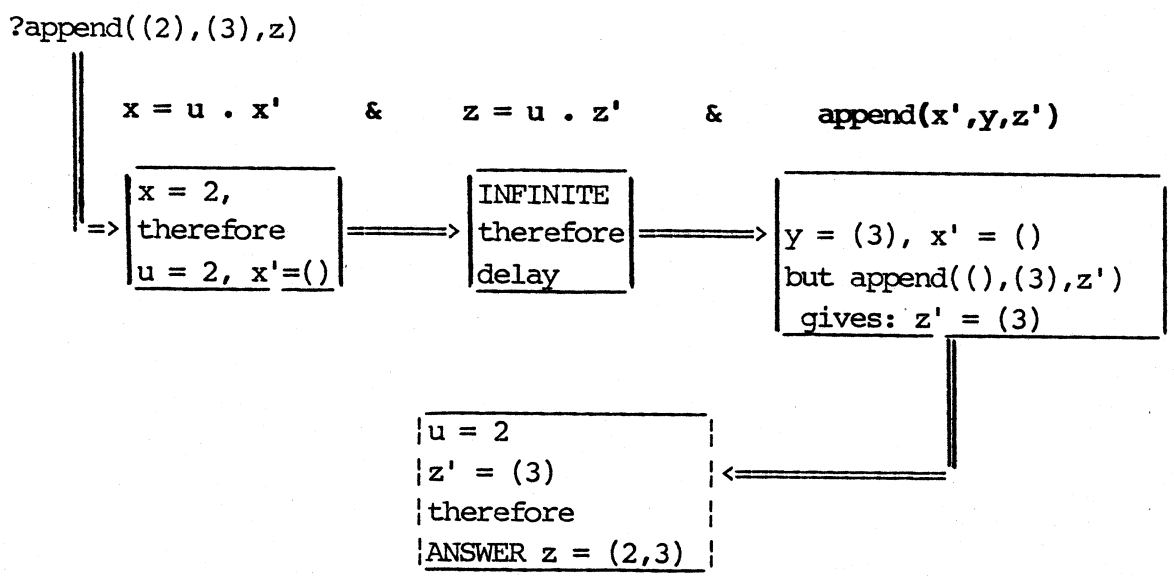
[7] WARREN D — Efficient Processing Of Interactive Relational Database Queries Expressed In Logic, Edinburgh DAI Research Paper No 156

[8] WARREN D & PEREIRA F — An Efficient Easily Adaptable System For Interpreting Natural Language Queries, Edinburgh DAI research paper no 155

457

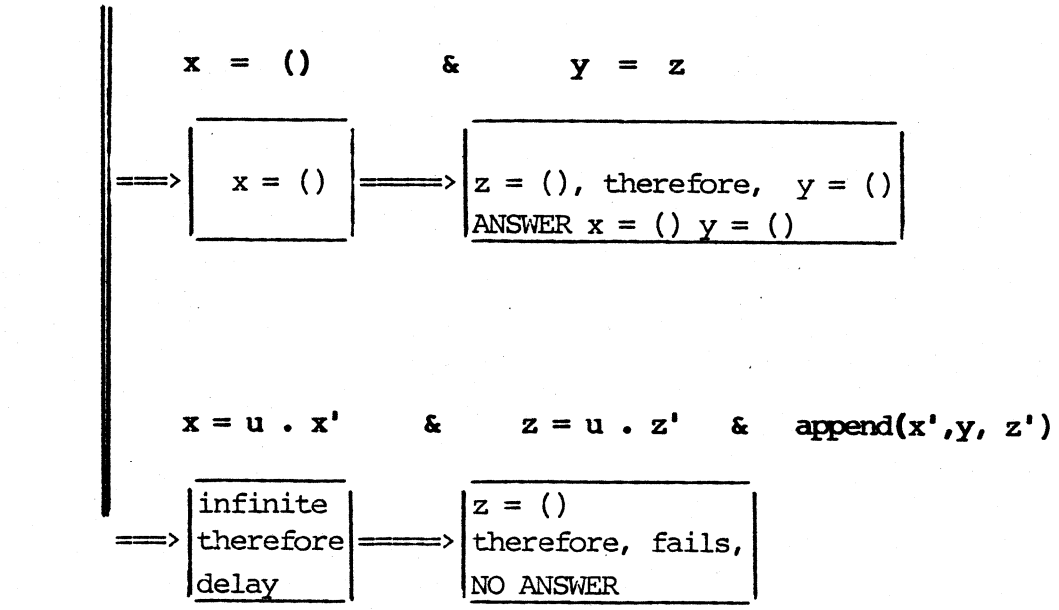
Appendix 1 Append forward

What is the result of appending (2) to (3)?



Appendix 2a Append backwards

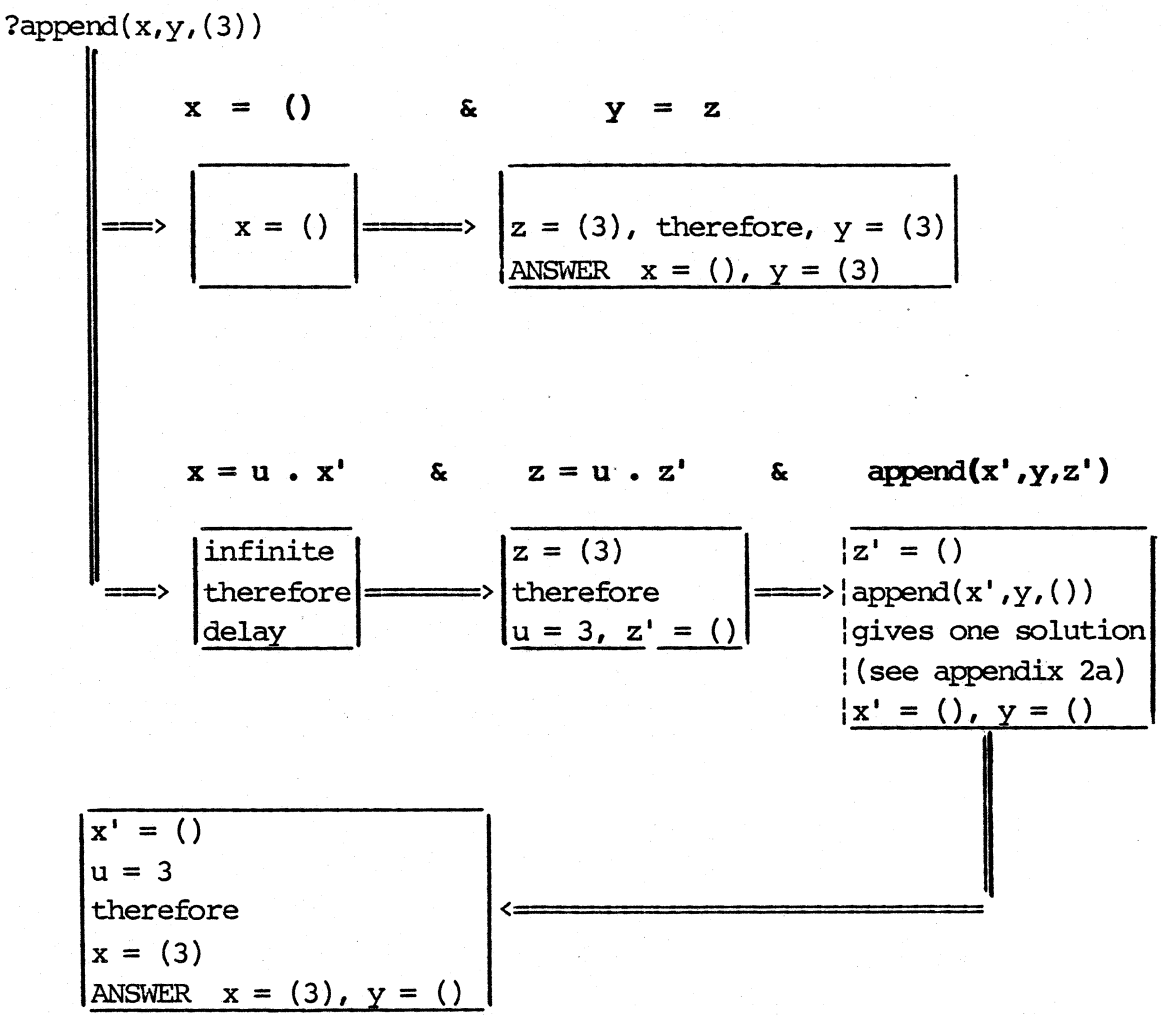
What two lists appended together give the empty list?



458

Appendix 2b

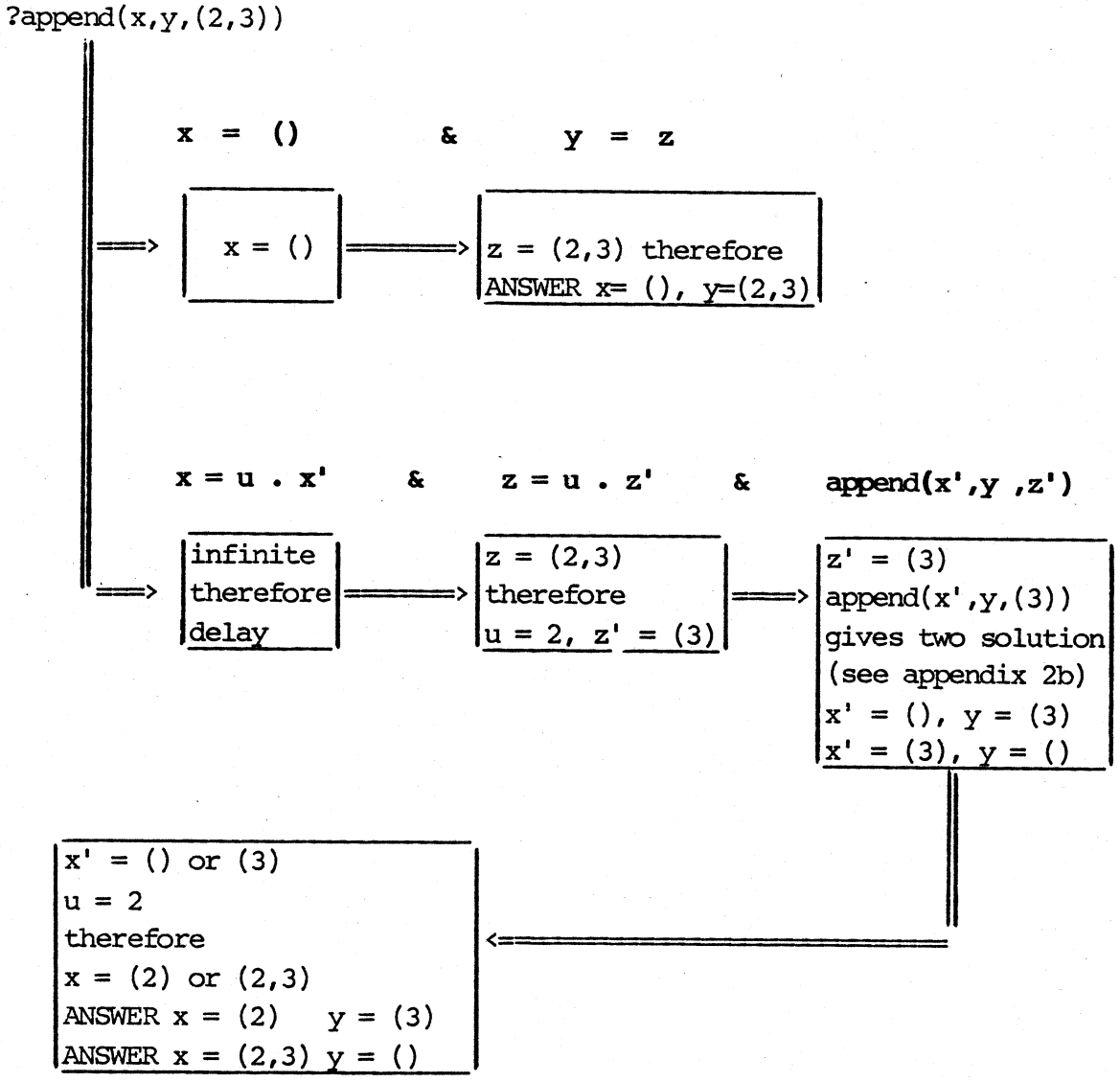
What two lists appended together give the list (3)?



459

Appendix 2c

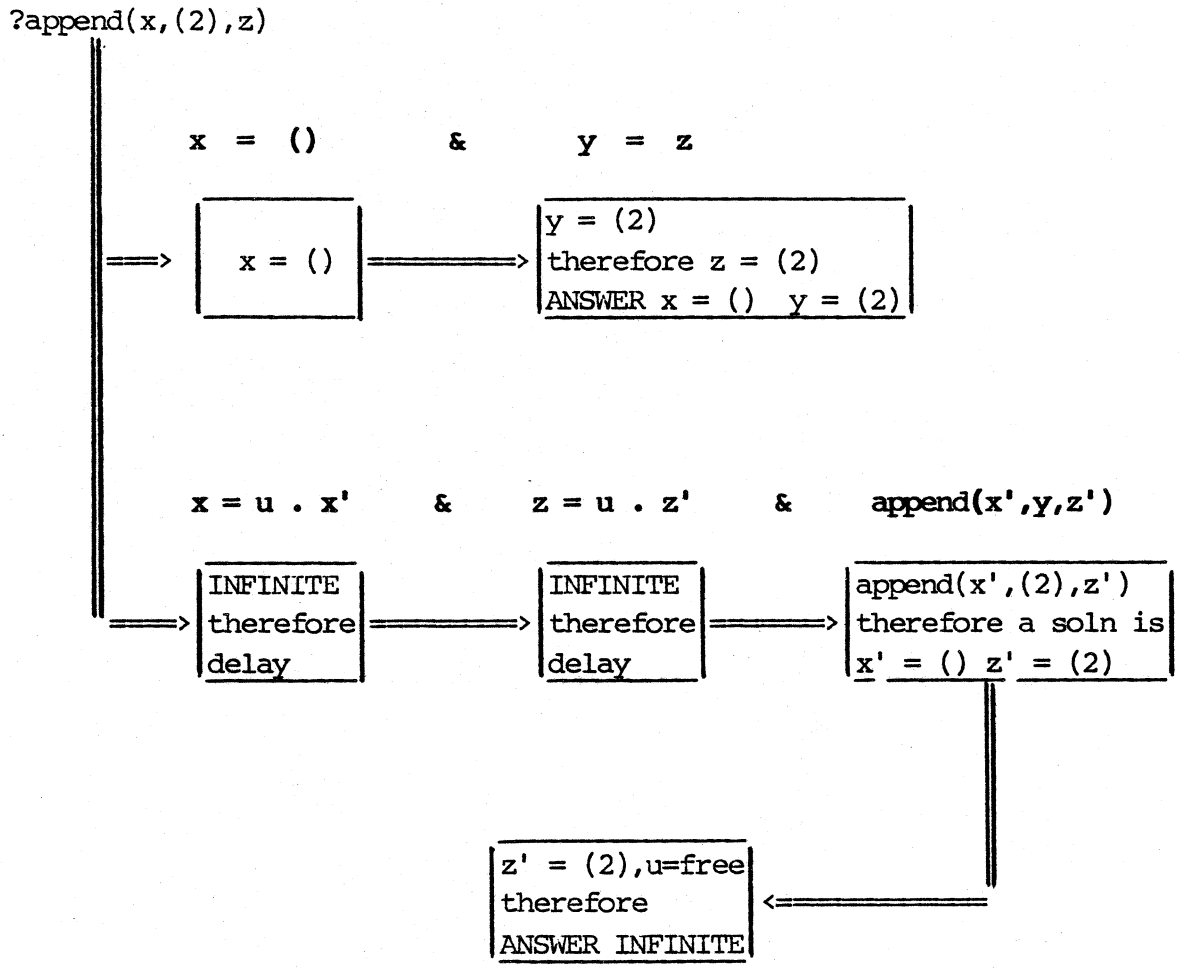
What two lists appended together give the list (2,3)?



460

Appendix 3 Append infinite

What are all the lists which end with a 2?



EB/SAC
SALLY836:EB 83/6

A NOTE ON COMPUTATIONAL COMPLEXITY OF LOGIC PROGRAMS

461

(Preliminary Draft)

Andrzej Lingas
Software Systems Research Center
Linköping University
S-581 83 Linköping, Sweden

Abstract

Shapiro defined three complexity measures over logic programs - goal-size, length and depth - and showed their relation to complexity measures for alternating Turing machines. We introduce the fourth complexity measure - conjunctive goal-size - and employing the known ideas of Turing-machine complexity theory we analyze the relation among the complexity measures over logic programs. In particular, for any deterministic logic program of conjunctive goal-size $S(n)$ and length $L(n)$ we can construct an equivalent deterministic logic program of depth $O(log(L(n))$ and length $O(L(n))$, and if the program is strongly deterministic then we can construct another equivalent strongly deterministic logic program of goal-size $O(log(S(n)) + log(L(n)))$ and length $O(S(n)L(n))$.

Introduction

The idea of procedural interpretation to Horn-clause logic begun a new era in logic programming. Today, the programming language Prolog, based on this idea, is a viewed as a start point to the basic programming language of the fifth generation computer systems [FGCS81].

The standard method of executing a program in Prolog is by so called backtracking, consuming a large amount of time and space. In order to achieve the planned speed up in time performance, Japaneses have to improve backtracking by mixing with other methods, for instance bottom-up, and work out an efficient parallel implementation of Prolog. A solid analysis of the computational complexity of logic programs should precede the speed up efforts.

In a large part our goal is to use the similarity between logic programs and alternating Turing machines in order to derive relationships among various complexity measures over logic programs. Efficient implementing of logic programs in various computational models may benefit from these results. As these results rephrase in part known facts from Turing machine theory in the language of logic programming, they seem to be of smaller importance for abstract complexity theory. The other our goal is to comment informally on the possibility of a fast parallel implementation of logic programs, and, on complexity of bottom-up computations of logic programs that are neglected in the logic programming society.

We totally adopt Shapiro's definitions of *definite clauses, goals, conjunctive goals,* clause's *head* and *body, logic program, goal reduction, substitution, unifier, derivation* and *refutation* of a goal from a logic program, the phrase "a program P *solves* a goal", *refutation tree, length, depth, goal-size* of refutation (see [Sh82a]).

The author came to the conclusion that it is natural and convenient to allow also variable free *initial axioms* as input data.

Initial axioms are inserted in the list of axioms of a logic program before starting its computation.

A pair (G, A) consisting of an initial goal G (possible a conjunctive goal) and a set of initial axioms is called an initial *goal-axiom pair.*

A goal-axiom pair (G, A) has a refutation from a logic program P if G has a refutation from $P \cup A$ in the Shapiro's sense.

The interpretation of a logic program P, $I(P)$, is the set of all variable-free goal-axiom pairs that are constructable from predicates, constants and functors appearing in the language in which P is written, and have a refutation from P.

Following our modification of logic program semantics in comparison with Shapiro, we redefine complexity measures over logic programs as follows:

A logic program P is respectively of *goal-size, depth, length complexity* $C(n)$ if for any goal-axiom pair in $I(P)$ of size n there respectively exists a refutation of goal-size, depth, length $C(n)$.

For the definitions of *non-deterministic* and *deterministic Turing machine* the reader is referred to [CKSh82].

Moreover we use the following definitions:

(1) An *axiom* is a clause with the empty body.

(2) Given a computation of a logic program, C, a *reduction step* of C is the reduction of a chosen goal to a sequence of new goals by a single application of a clause in the program.

(3) Let P, R be a logic program and a refutation, respectively.

The *conjunctive goal-size* of R is the maximum size of the current list of goals at any reduction step of R (respectively, goal size is the maximum size of any unit goal at any step of R). P is of *conjunctive goal-size complexity* $U(n)$ if for any goal-axiom pair G in $I(P)$ of size n there is a refutation of G from P of conjunctive goal-size $\leq U(n)$.

The non-deterministic length of R is the number of nodes in the refutation tree

such that there are at least two clauses whose heads match the goal chosen to reduce. P is of non-deterministic length complexity $N(n)$ if for any goal-axiom pair G in $I(P)$ of size n there is a refutation of G from P of non-deterministic length $\leq N(n)$.
If $N(n) \equiv 0$ then P is *strongly deterministic.*
If every goal-axiom pair in $I(P)$ admits only one refutation then P is *deterministic,* see [H81].
Obviously, if P is strongly deterministic then it is deterministic. The notion of strong determinism for logic programs corresponds to that of determinism for Turing machines.
(4) We assume a standard list representation. The term [] denotes the empty list, and the term $[X|Y]$ stands for a list whose head is X and tail is Y. A string $\alpha_1\alpha_2...\alpha_n$ is represented by the list $[\alpha_1|[\alpha_2|[...|\alpha_n]]]$. With the exception of Theorem 3 integers n are represented as n-fold composition of the functor s applied to the constant 0. Writing a logic program, we skip the clauses and axioms defining the arithmetic predicates of $=, <, \leq, >, \geq$. Finally, we assume that we can test equality between an atom and term by applying a standard equality and inequality predicates built in the formalism of logic programs.
(5) According to the assumed string representation (see 4), Turing machine M is equivalent to a logic program P if after erasing the square brackets and the symbol "|" in the words of $L(M)$, we obtain $I(P)$.

Relationships among Complexity Measures over Logic Programs

In the following remark, we can find a couple of obvious observations on complexity measures over logic programs.

Remark 1. Let P be a logic program of depth complexity $D(n)$, conjunctive goal-size complexity $G(n)$ and length complexity $L(n)$. The following inequalities hold:

$D(n) \leq L(n)$
$L(n) \leq d^{G(n)}$ where d is a constant uniform in P.
Moreover, if we restrict initial goals to single goals then, we have $L(n) \leq c^{D(n)+1} - 1$ where c is the maximum number of goals in a clause of P.

In several computational models, the depth complexity is a natural lower bound on the time taken by parallel evaluation. In the computational model of logic programming it is hard to approximate the lower bound with efficient parallel computations. Simply, solving a conjunctive goal with shared variables

cannot be spawned directly.

By virtue of the following theorem, for any logic program there exists an equivalent logic program of fairly small depth. The proof is by applying Savitch's trick, originally applied to simulate non-deterministic space bounded Turing machines by deterministic ones (see [Sa70]), and then, by time bounded alternating Turing machines [CKS80].

Theorem 1. Any logic program P of length complexity $L(n)$ and non-deterministic length complexity $N(n)$ can be transformed into a logic program Q such that a goal-axiom pair $((G_1, G_2, ..., G_l), (A_1, A_2, ..., A_k))$ of size n is in $I(P)$ if and only if there exists a refutation of the corresponding goal-axiom pair $(p([G_1|[G_2|[...|G_l]]]$, [], $\lceil log(L(n))\rceil$, $(p([A_1|X], [X], 0)$, $p([A_2|X], [X], 0)$, ..., $p([A_k|X], [X], 0)))$ from Q of depth $\lceil log(L(n))\rceil$, length $4L(n)$ and non-deterministic length $N(n)$. If the program P is deterministic (respectively, strongly deterministic) then Q is also deterministic (respectively, strongly deterministic).

Proof. To form the clauses of Q, we use only the predicate $p(X, Y, i)$. It reads:

If X and Y are lists representing goals and i is a natural number then the goals from X can be reduced to those from Y in 2^i reduction steps.

For each clause $A \leftarrow B_1, ..., B_k.$ of P, the program Q contains the axiom $p([A|X], [B_1|[B_2|...[B_k|X]]], 0)$. Note that the predicates from P become functors here. Next, Q contains the axiom $p([], [], 0)$. saying that we can reduce the empty list of goals to itself in one reduction step. The only clause with non-empty body in Q is as follows:

$$p(X, Y, s(j)) \leftarrow p(X, Z, j), p(Z, Y, j).$$

Given a refutation of G from P, of length $L(n)$, there exists a refutation of G from P, say R, such that at each reduction step in R the first clause on the current list of goals is chosen to reduce and R is of length $L(n)$. Having R, we form a refutation of the corresponding initial goal from Q by applying the only clause of Q in depth-first manner, and then, the axioms of Q. As a result, we obtain a refutation whose tree has leaves labelled by instantiated predicate $p(X, Y, 0)$ corresponding to single reduction steps of R. The length of the refutation does not exceed $2^{\lceil log(L(n))+1\rceil} - 1$. Its non-deterministic length is the same as that of R. If R is the only refutation of the initial goal-axiom pair from P then it is the only refutation of the corresponding initial goal-axiom pair from Q.
Conversely, given a refutation of the corresponding goal from Q, we can easily find out a refutation of G from P. ∎

In Savitch's simulation of non-deterministic space bounded Turing machines with deterministic space bounded Turing machines, the intermediate tape configuration (corresponding to the intermediate list of goals Z in the above proof) is determined by exhaustive search (see [Sa70]). In the proof of Theorem 1, the intermediate list of goals substituted for Z is the outcome of calling $p(X, Z, j)$ (in Concurrent Prolog [Sh82b], the basic clause in Q would be rather written as $p(X, Y, s(j)) \leftarrow p(X, Z, j), p(Z?, Y, j)$). From the point of deterministic simulation, our method of finding the intermediate state is more efficient than Savitch's one if the non-deterministic length complexity of P is small, and worse otherwise.
The first who showed how to simulate Turing machines with logic programs was Tarlund [T67]. Shapiro proved a close relationship between complexity of alternating Turing machines and complexity of logic programs [Sh82a]. The following theorem reveals relationships between complexity of non-deterministic Turing machines and complexity of logic programs (In this theorem, as well as in Theorem 3 and Corollary 1 and 2 we informally use the notion of simulation whose meaning can be deduced from the proof of Theorem 3).

Theorem 2. Any multi tape (deterministic) Turing machine operating in time $T(n)$, and space $S(n)$ can be simulated by a (strongly deterministic, respectively) logic program of length complexity $O(T(n))$, and conjunctive goal-size complexity $O(S(n))$. Conversely, any (strongly deterministic) logic program of length complexity $L(n)$, and conjunctive goal-size complexity $S(n)$ can be transformed into an equivalent (deterministic, respectively) Turing machine operating in time $O(L(n) \times S(n)^2)$, and space $O(S(n))$.
Hint. Note that a single reduction step can be simulated by a deterministic Turing machine in time $O(S(n)^2)$ (see [R65]) and read the proof of Theorem 4.4 and 5.4 in [Sh82a]. ∎

By Theorem 1 and 2 we obtain the following corollary:

Corollary 1. Any (deterministic) Turing machine operating in time $T(n)$, and space $S(n)$ can be simulated by a (strongly deterministic, respectively) logic program of depth complexity $O(log(T(n))$, length complexity $O(T(n))$, and conjunctive goal-size complexity $O(S(n))$.

Probably, several important problems solvable by deterministic Turing machines in polynomial time are not solvable in parallel time $O(log^k n)$, i.e. by parallel machines with polynomial number of processors with fixed fan-in and fan-out, running in time $O(log^k n)$ (see [B77],[CKS81]). As by Corollary 1,

deterministic Turing machines operating in polynomial time can be simulated by deterministic logic programs of logarithmic depth complexity, probably a small depth complexity of a logic program does not ensure the existence of a fast parallel implementation of the program, in the general case. It seems that the requirements that a logic program should satisfy to admit an essential parallel speed up are more complex. In the next section, we shall briefly discuss this problem from the point of view of bottom-up computations. Here, we informally propose the following requirements, coherent with the top-down nature of derivations from logic programs.

Let P be a logic program of length complexity $L(n)$. For i, j, let $R_{i,j}(n)$ be the equivalence relation between conjunctive goals such that $G_1 R_{i,j}(n) G_2$ if and only if for any goal-axiom pair of size n, G, any refutation of G from P with the the i —th element G_1 performs the same i-th through j-th reduction steps as any refutation of G from P with the i-th element G_2. In other words, to determine the i-th through j-th reduction steps of a refutation of G from P whose i-th element is G_1 it is sufficient to know a representative of the equivalence class of $R_{i,j}(n)$ for G_1. Suppose that for $n \in N$ there exists a tree T_n of fixed degree with leaves consecutively labeled by 1 through $L(n)$, and a number m_n such that for any subtree of T_n with the leftmost leaf labelled by i and the rightmost leaf labelled by j, the number of equivalence classes of $R_{i,j}(n)$ is at most m_n. In the simplest case, the tree T_n may correspond to the refutation tree of P. Given an goal-axiom pair of size n, G, we can recursively find a refutation of G from P (if it exists) by applying divide and conquer strategy induced by T_n and trying all representatives of the equivalence classes of $R_{i,j}(n)$ in parallel. Provided that T_n and the representatives are given, the refutation can be determined in time $O(log(m_n) \times height(T_n))$ with the use of $O(2^{log(m_n) \times height(T_n)})$ processors. In particular, if m_n is a constant uniform in n and $height(T_n) = O(logn)$, P can be implemented in parallel time $O(logn)$. The reader can find more details about this approach, expressed rather in terms of Turing machines, in [L83]. Here, we offer only the following simple example.

Example 1

Let us consider the following logic program, $delmem(Z, X, Z', H)$, where Z is an input linear list of a constant length over a finite alphabet Σ, X is an input list over Σ organized as a complete binary tree of height H, Z' is the output list composed of all the elements of Z that are not in X, $member(A, Z)$, $notmember(A, Z)$, $delete(A, Z, Z')$ stand for the standard predicates testing membership of A in Z and deleting A from Z (i.e. $Z' = Z - A$) respectively (see

[CM81]), we may assume without loss of generality these standard predicates to be available primitives since they are applied to sublists of the input list Z which is of fixed length in this example.

$$delmem(Z, X, Z', H) \leftarrow delmem(Z, X, Z', 0, H).$$

$$delmem(Z, [X \mid Y], Z', K, H) \leftarrow$$

$$K < H, delmem(Z, X, Z'', s(K), H), delmem(Z'', Y, Z', s(K), H).$$

$$delmem(Z, A, Z', H, H) \leftarrow member(A, Z), del(A, Z, Z').$$

$$delmem(Z, A, Z, H, H) \leftarrow notmember(A, Z).$$

Note that if we neglect labels, the form of a refutation tree of P for any goal-axiom pair with the input list X of length n is totally determined by n. Let T_n be a tree of such a form, with leaves consecutively labelled by 1 through n. Clearly, if i and j are the labels of the rightmost and the leftmost leaf in a subtree of T_n, then the i-th through j-th reduction steps in any refutation of an goal-axiom pair with the input list X of length n from our logic program is a refutation of the goal $delmem(U, Y, U', k, h)$ corresponding to the root of the subtree. Thus, the i-th through j-th steps are totally determined by the instantiation of Z, U. Therefore, the equivalence classes of the relation $R_{i,j}(n)$ can be identified with the possible instantiations of Z. As the input list Z is of a fixed length, the number of possible instantiations of Z is a constant uniform in n. Hence, the number m_n is a constant uniform in n here. It is not difficult to see that the language specified by $delmem(Z, X, Z', K, H)$ is regular but in the general case, the language specified by a logic program for which $m_n = O(1)$ is not necessarily regular [L83]. The tree T_n induces the same divide and conquer strategy as the recursive definition of $delmem(Z, Y, Z', k, h)$, therefore, we do not need to transform P in this respect. To try all of the representatives of equivalence classes of the relations $R_{i,j}(n)$, equivalently all possible instantiations of U, it is sufficient to add the following clauses with B ranging over all possible instantiations of Z:

$$delmen(Z, [X \mid Y], Z', K, H) \leftarrow$$

$$K < H, delmem(Z, X, B, s(K), H), delmem(B, Y, Z', s(K), H).$$

It is easy to see that by fully using the OR-parallelism introduced by the above, additional clauses, $delmem(Z, Y, Z', k, h)$ can be implemented in parallel time $O(logn)$.∎

The following theorem relates time and space complexity of Turing machines to goal-size complexity of logic programs. The proof is analogous to the proof of Chandra *et al.* showing $\bigcup_{c>0} DTIME(c^{S(n)}) \subseteq ASPACE(S(n))$ [CKSh82].

Theorem 3. Let M be a deterministic Turing machine operating in time $T(n)$ and space $S(n)$. M can be simulated by a strongly deterministic logic program Q of goal-size complexity $O(logT(n) + logS(n))$.

Outline of Proof. We assume several restrictions on M following the proof of Theorem 3.4 in [CKS80]. In particular, M has only one tape, on the tape the input word is written, M accepts an input by entering its unique accepting state q_A with the head scanning the $T(n) + 1st$ tape square, etc. (see [CKS80] for details). A computation of M on the input word is described as a sequence of configurations, each in the form $\alpha q \beta$ where $\alpha\beta$ describes the contents of squares 0 through $4T(n)$, q is the current state of M and the head of M points the rightmost symbol of α.

For any four symbols from the tape alphabet and the set of states of M, $\delta_{-1}, \delta_0, \delta_1, \delta_2$, among which at most one represents state of M, there is unique symbol δ such that for any j if $\delta_{-1}, \delta_0, \delta_1, \delta_2$ occupy positions $j-1, j, j+1, j+2$ in a configuration of M, then δ occupies the position j in the next configuration of M. For each such quintuple $\delta_{-1}, \delta_0, \delta_1, \delta_2, \delta$, the program Q contains the axiom $next(\delta_{-1}, \delta_0, \delta_1, \delta_2, \delta)$. . The basic predicate in Q is $accept(j, t, a)$. It says that in the t — th configuration of the computation of M, the j — th square contains the symbol a.

To prove the theorem we cannot represent the integers j, t using the unary notation defined in the previous section. Here the integer of binary representation $b_1, ..., b_l$ is written as $[b_l|[...|b_1]]$. The successor predicate is defined as follows:

$$suc([1|X], [0|Y]) \leftarrow suc(X, Y).$$

$$suc([0|X], [1|X]).$$

$$suc([], 1).$$

The definitions of the predicates of $<$ and $\leq$ for this specific representation of integers are left to the reader.

The main clause in Q is as follows:

$$accept(j, u, X) \leftarrow suc(t, u),\ suc(i, j),\ accept(i, t, Y),$$

$$accept(j, t, Z),$$

$$suc(j,k),\ accept(k,t,W),$$

$$suc(k,l),\ accept(l,t,T),$$

$$next(Y,Z,W,T).$$

To verify the initial contents of the tape we use the clauses $accept(j,0,X) \leftarrow input(j,X)$. The integers occurring in any derivation of $accept(T(n)+1, T(n), q_A)$ from P have binary representation of the length not exceeding $\lceil min\{log(T(n)), log(S(n))\}\rceil$. To prove the theorem, we show by induction that $accept(j,t,a)$ can be proved in $O(L(n))$ reduction steps if and only if the given interpretation of $accept(j,t,a)$ is right. ∎

By Theorem 2 and 3 we obtain the following corollary:

Corollary 2. Any strongly deterministic logic program of length complexity $L(n)$ and conjunctive goal-size complexity $S(n)$ can be simulated by a strongly deterministic logic program of length complexity $O(L(n) \times S(n)^2)$, and goal-size complexity $O(log(L(n)) + log(S(n)))$.

Bottom — up Computations of Logic Programs and their Complexity

That what we mean by a computation of a logic program P might be specified as a *top-down computation* of P. A *bottom-up computation* of P is a reversed (top-down) computation of P, and can be briefly described as follows.

The computation starts from a set of instantiated axioms. At each step we non-deterministically pick a clause of P, $A \leftarrow B_1', B_2', ..., B_k'$ (it might be an axiom, i.e. $k = 0$). Then we non-deterministically choose a sequence $B_1, B_2, ..., B_k$ from the list of current axioms in order to unify it with the body of the previously chosen clause. The unification is via a substitution θ and the axiom $A\theta$. is added to the current list of axioms. The computation terminates when there exists a substitution θ unifying each initial goal with a member of the current list of axioms. The definitions of of *derivation, refutation* of an initial goal-axiom pair from P etc. as well as the definitions of *depth, length* and *conjunctive goal-size* complexity for bottom-up computations are similar to those for top-down computations, and are left to the reader.

Remark 2. If a logic program is of bottom-up depth complexity $D(n)$, bottom-up length complexity $L(n)$, bottom-up goal-size complexity $U(n)$, then it is of depth complexity $D(n)$, length complexity $L(n)$ and goal-size complexity $U(n)$.

Choosing a clause at a step of a bottom-up computation of P and a sequence of some current axioms in order to unify with the body of the clause, we do not know whether it leads to a proof of the initial goals. Moreover, the number of possible choices of the sequence of some current axioms may be of order n^k where k is the number of goals in the body of the chosen clause. That is why programs in Prolog are executed in a top-down manner. We may argue that if the program P is non-deterministic then choosing a clause in a top-down computation in order to unify its head with the selected goal, we neither know whether it will solve the goal. However, if we do not loop then we may backtrack in case of failure like the running Prolog interpreters whereas the definition of failure for a bottom-up computation is not clear. Neverthless, it is author's feeling that for an important class of logic programs bottom-up computations are essentially more efficient than (top-down) computations. This class may include so called *dynamic programming* procedures which recursively generate a lot of symmetric subgoals in order to solve the original goal.

An example of a logic program for the dynamic programming procedure of Cocke, Kasami and Young, accepting words from the language $L(G)$ where $G = (N, \Sigma, P, S)$ is a context-free grammar in Chomsky normal form (see [AHU74]), is shown as Program 1.

Program 1

$p_A(i,j) \leftarrow q_A(i,i,j).$ for $A \in N$,
$q_A(i,k,j) \leftarrow s(s(k)) < j, q_A(i,s(k),j).$ for $A \in N$,
$q_A(i,k,j) \leftarrow i < k < j, p_B(i,k), p_C(k,j).$ for $A \rightarrow BC \in P$,
$p_A(i,s(i)) \leftarrow i < n, input(a,i).$ for $A \rightarrow a \in P$.

The program is design to succeed on the goal-axiom pair consisting of the goal $p_S(0,n)$ and the axioms $input(w_i,i).$ where $w_1, ...w_n$ is the input word if and only if the input word belongs to $L(G)$. In the worst case we may have to backtrack an exponential in n number of times in order to find a (top-down) computation accepting w whereas a bottom-up computation yields an answer in $O(n^3)$ deduction steps if it proves a new goal at each deduction step. Why are bottom-up computations successful here ? Simply, there are only $O(n^3)$ variable free goals that can be solved by Program 1 starting from the initial axioms.

Definition 1. Let R be a refutation. The *goal number* of R is the total number of distinct goals in the nodes of the refutation tree. A logic program P is of *goal number complexity* $G(n)$ if for any goal A in $I(P)$ of size n there exists a refutation of A from P of goal number $\leq G(n)$.

Our observation about bottom-up computations of Program 1 can be generalized as follows:

Remark 3. Let P be a logic program. P is of goal-number complexity $G(n)$ if and only if it is of bottom-up length complexity $G(n)$. Moreover, if P is of goal-size complexity $U(n)$ then it is of goal number complexity $d^{U(n)}$ where d is a constant uniform in P.

The analogous remark for (top-down) computations would not be true. Simply, it might happen that to solve a given goal, we have to solve the same goal several times. In implementing (top-down) computations we can get rid of the above inefficiency by dynamically extending the original set of axioms of P by the solved intermediate goals.

By Theorem 2 and Remark 2 and 3, we can observe that any Turing machine operating in polynomial time can be simulated by a logic program of polynomial goal number complexity.

In a simple parallel implementation of bottom-up computations, we do not encounter the problem of variable sharing for subgoals of equal rank. Therefore, the depth complexity and the time taken by a single deduction step seem to decide about the time performance of a bottom-up computation of a logic program of polynomial goal number complexity, in a parallel computational model. The recent paper of Lewis and Statman [LS8?] has shown the problem of unification between first order terms to be complete in co-NLog Space. Therefore, the existence of a parallel algorithm for the unification problem operating in time $O(log n^k)$ and using a polynomial in n number of processors would imply the existence of such algorithms for any problem from NLog Space or co-NLog Space, which seems unlikely. Hence, we cannot count on a parallel implementation of the single deduction or reduction step in time $O(log n^k)$. If the logic program P in Theorem 1 is not patological then the bottom-up depth complexity of the resulting program Q is equal to the depth complexity of Q. Therefore, by Remark 2, we can usually apply Theorem 1 in order to compress the bottom-up depth complexity. The following theorem, analogous to Theorem 1, shows how to achieve this for any logic program.

Theorem 4. Let P be a logic program of bottom-up length complexity $L(n)$. Let $B_1, ..., B_m$ be the list of all axioms in P. P can be transformed into a logic program Q such that a goal-axiom pair $((G_1, ..., G_k), (A_1, ..., A_l))$ is in $I(P)$ if and only if the corresponding goal-axiom pair $(p([B_1|[...|B_m]], [G_1|[...[G_k|—]]], \lceil log(L(n))\rceil), (p([X], [A_1|X], 0), ..., p([X], [A_l|X], 0)))$ has a refutation from Q of depth $\lceil log(L(n))\rceil + \lceil log(n + L(n))\rceil$.

Outline of Proof. The proof is again by applying Savitch's trick, analogously as in the proof of Theorem 1. Here the predicate $p(X, Y, i)$ says:

If X and Y are lists of axioms and i is natural number then the axioms in Y can be derived from those in X in 2^i (bottom-up) deduction steps.

The axioms chosen from the current list of axioms in order to unify with the body of chosen clause may occupy various positions on the list. Therefore, we include in Q the following clauses to pull the chosen axioms to the front of the list (because the list of axioms may be of length $n + L(n)$ we again apply Savitch's trick).

$$p(X, Y, 0) \leftarrow q(X, Y, \lceil log(n + L(n)) \rceil).$$

$$q(X, Y, s(j)) \leftarrow q(X, Z, j), q(Z, Y, j).$$

$$q([X|[Y|Z]], [Y|[X|Z]], 0) \leftarrow X \neq Y.$$

$$q(X, X, 0).$$

Finally, for each clause $A \leftarrow B_1, ..., B_n$ in P we have the corresponding axiom $q([B_1|[...[B_n|X]...], [A|[B_1|[...|[B_n|X]...], 0)$. ∎

In the above theorem, the program Q is non-deterministic even if the program P is strongly deterministic (compare with Theorem 1).

Possible Extensions

(1) The goal-size complexity of Q in Theorem 1 might be as large as $L(n) \times U(n)$ if P is of goal-size complexity $U(n)$. It seems possible to generalize Theorem 1 by showing a trade off between the depth complexity and the goal-size complexity of Q.
(2) It is possible to formalize the notion of simulation or introduce a more general concept of equivalence among logic programs and Turing machines.
(3) It would be interesting to design a parallel algorithm for the unification problem operating in time $O(n^\alpha)$ and using (n^β) processors where $\alpha < 1$ and $\beta < 1$.

Acknowledgements

I would like to express my appreciation to Jan Komorowski, and Jan Maluszynski for their encouragment and remarks.

References

[AHU74] Aho, A.V., J.E. Hopcroft, and J.D. Ullman, *The design and Analysis of Computer Algorithms,* Addison-Wesley, Reading, Mass., 1974.

[B77] Borodin, A.B., *On relating time and space to size and depth,* SIAM J. Compt., vol. 6(4), 1977.

[CKS81] Chandra,A.K., D.C. Kozen, L.J. Stockmeyer, *Alternation,* Journal of the ACM 28(1), 1981.

[CM81] Clocksin, W.F. and C.S. Mellish, *Programming in Prolog,* Springer-Verlag, 1981.

[H81] Hogger, C.J., Derivation of Logic Programs, Journal of the ACM 28(1), 1981.

[FGC81] Proceedings of International Conference on Fifth Generation Computer Systems, Tokyo, 1981.

[L83] Lingas, A., *Languages with Sparse Computations,* in preparation.

[LS8?] Lewis, H.R. and R. Statman, *Unifiability is Complete for co-NLog Space,* unpublished manuscript, 198?.

[R65] Robinson, J.A., *A machine oriented logic based on the reduction principle,* Journal of the ACM 12, January, 1965.

[Sa70] Savitch, W.J., *Relationships Between Non-deterministic and Deterministic Tape Complexities,* JCSS 4(2), 1982.

[Sh82a] Shapiro,E.Y., *Alternation and the Computational Complexity of Logic Programs,* International Conference on Logic Programming, Marseil, 1982. [Sh82b] Shapiro,E.Y., *A Subset of Concurrent Prolog and Its Interpreter,* unpublished manuscript, 1982.

[T77] Tarlund, S.A., *Horn Clause Computability,* BIT 17, 1977.

ON THE FIXED-POINT SEMANTICS OF HORN CLAUSES WITH INFINITE TERMS

M.Falaschi, G.Levi, C.Palamidessi
Dipartimento di Informatica
Universita' di Pisa, Italy

1. INTRODUCTION

Infinite terms (streams) have been introduced in several PROLOG-like languages [2,3,4,8,10] in order to define parallel communicating processes. The resulting operational semantics is quite similar to Kahn-McQueen's model [5], characterized by agents which communicate through channels. Most of the above mentioned languages are annotated versions of PROLOG. Hence some of the most relevant features of PROLOG, such as the ability to define relations, get lost.

If infinite terms are added to pure PROLOG (i.e. Horn clauses), the definition of a "good" fixed-point semantics is still an open problem. In [1] a greatest fixed-point construction is proposed. Such solution, however, is not satisfactory, because:

i) the greatest fixed-point semantics gives a non-empty denotation not only to nonterminating procedures which compute infinite terms, but also to "bad" standard nonterminating programs;
ii) the construction is not always effective, i.e. there exist programs whose greatest fixed-point cannot be computed.

In this paper we propose two semantics based on a least fixed point construction. In the first semantics we only consider all the finite approximations of an infinite term, while the second semantics allows to handle infinite terms. The language we will consider is a many sorted version of PROLOG. Its syntax will be defined in the next section. It is worth noting that the sorting mechanism will allow us to distinguish finite and infinite terms.

2. SYNTAX AND DERIVATION RULE

The language alphabet is composed by:

1) A set S of identifiers for the representation of the sorts. A sort s is:

a) simple if s belongs to S. The set of simple sorts is partitioned into two disjoint classes, canonical and non-canonical sorts, to cope with finite and infinite data structures respectively.
b) functional if s belongs to S* --> S . If s has the form: $s_1 \times \ldots \times s_n$ --> s', and at least one of the s_i's is non-canonical, then s' is non-canonical too.
c) relational if s belongs to S .

2) A family C of sets of constant symbols indexed by simple sorts. If s is a non-canonical sort, then the set of constants of sort s contains the special symbol ω_s, which denotes an undefined (not yet evaluated) data structure.
3) A family D of sets of data constructor symbols indexed by functional sorts.
4) A family V of numerable sets of variable symbols indexed by simple sorts.
5) A family R of sets of predicate symbols indexed by relational sorts.

The language data structures are obtained by applying data constructors to variables and constants of suitable sorts. More precisely, a term of sort s is:

i) a constant symbol of sort s.
ii) a variable symbol of sort s.
iii) a data constructor application $d(t_1,\ldots,t_n)$ such that $t_1,\ldots,t_n$ are data terms of sorts $s_1,\ldots,s_n$ and d belongs to D and has sort $s_1 \times \ldots \times s_n$ --> s.

A term which contains at least one occurrence of an undefined constant symbol is called suspension and denotes a not completely evaluated data structure. Because of the condition in 1.b), if one of the t_i's has a non-canonical sort (briefly is non-canonical), then also the term is non-canonical. In fact, the result of the application of a data constructor to its components (arguments) is a suspension if some of its components are suspensions.

The language basic construct is the atomic formula. An atomic formula is a predicate application $P(t_1,\ldots,t_n)$ such that $t_1,\ldots,t_n$ are data terms of sort $s_1,\ldots,s_n$ respectively, and P is a predicate symbol of sort $s_1 \times \ldots \times s_n$.

A set of atomic formulas can be interpreted as a collection of processes or agents [2,7] connected by channels. Each atomic formula denotes a process. There exists a channel connecting processes P_h and P_k, if there exists a variable symbol which occurs in the atomic formulas denoting P_h and P_k. The basic activity is message passing through channels and reconfiguration of the collection of processes. Informations can pass through a channel in both directions. This is not the case of the SCA model [7], as well as of the Kahn-McQueen model [5].

The dynamic behaviour of the collection of processes is specified by a set of clauses, which are expressions of the language defined as follows:

1) A definite clause is a formula of the form:

$$A \leftarrow B_1, \ldots, B_n$$

where A and the B_i's are atomic formulas. If n=0 the clause is called "unit clause" and is denoted as follows:

$$A \leftarrow \lambda$$

All the variables occurring in a clause are viewed as universally quantified.

2) A negative clause (goal statement) is a formula of the form:

$$\leftarrow A_1, \ldots, A_m$$

where the A_i's are atomic formulas. If m=0 it is a null clause denoted by

$$\leftarrow \lambda \quad (\text{or } \square)$$

From a logical viewpoint, the symbol "," denotes the logical connective AND, the symbol "<--" denotes the logical implication, and λ is the neutral element with respect to the operator ",", that is $\leftarrow A, \lambda = \leftarrow A$

The notion of derivation of a goal statement from a given goal statement and a program is essentially the same defined for PROLOG [6], and is based on resolution [9]. The only trivial difference has to do with sort checking.
The relation

$$G \; |\!\xrightarrow[W]{\theta}\!> G'$$

denotes that the goal statement <-- G' is derivable from the goal statement <-- G and the program W, with the substitution θ, which is the composition of all the substitutions used in the elementary derivations.

If, for some θ, the relation

$$G \; |\!\xrightarrow[W]{\theta}\!> \lambda$$

holds, then <-- G is refutable in W.

Our interpretation of goal statements and clauses is exactly the same given by Kowalski [6] for PROLOG. However, we think of a goal statement as denoting a collection of processes. The derivation of a new goal statement corresponds to a reconfiguration of the collection. Each elementary variable binding in a unification can be seen as a message passing from a producer to a consumer. Our interpretation is motivated by the fact that we view processes as non terminating procedures which produce (or consume) infinite data structures. Such procedures have an empty denotation in PROLOG, both from the operational and the fixed-point semantics viewpoint.

3. OPERATIONAL SEMANTICS

In standard Horn Clause Logic the concept of computation of a goal statement is essentially based on the refutation of that goal statement, (i.e. the derivation of the null clause), and therefore on the concept of termination. In other words, the result of a computation of a goal statement (i.e. its operational semantics) is the relation established, for each predicate in the goal, by the substitutions determined in all the possible refutations [6].

This definition of operational semantics results inadequate to describe processes which handle infinite terms (streams). Consider, for example, the following program:

W = {list(x,x.L) <-- list(s(x),L)}

where the sort of x is 'naturals' (canonical sort), the sort of L is 'streams of natural' (non canonical sort), '.' denotes the stream of naturals constructor, and 's' denotes the successor constructor on naturals (for the sake of simplicity we will use 1 instead of s(0), 2 instead of s(s(0)), etc.).

Since the goal statement <-- list(0,L) has no refutations in W, the denotation of the predicate list given by the standard operational semantics is an empty relation. In spite of this, a derivation of list(0,L) produces, step by step, the substitutions:

L = 0.L
L = 0.1.L
L = 0.1.2.L etc...

It is easy to see that an infinite computation of this goal statement will lead L to be instanced to the infinite list of natural numbers. In general every process which produces infinite terms has the same problems with respect to its semantics definition, since its computation necessarily does not terminate.

The solution we propose is based on the introduction for each predicate symbol P which is non-canonical (i.e. which handles infinite terms), of a <u>terminal clause</u> (unit clause) defined as follows:
If P has sort $s_1 \times \ldots \times s_n$, then the terminal clause has the form $P(t_1,\ldots,t_n)$ <-- , where each t_i is:
- a variable of sort s_i, if s_i is canonical
- the undefined constant symbol ω_{s_i}, if s_i is non-canonical.

The terminal clause is added only if there exists no unit clause, in the program, for which there is a superposition. This condition is necessary because it must not be possible to introduce new solutions by adding a terminal clause. The new terminal clause must only allow termination.

Note that if there exists a terminal clause, for which there exists a superposition with the new one, then it contains some non-canonical terms that can be substituted with ω. For this reason the termination is guaranteed in this case.

In our example the terminal clause is

$$list(n,\omega) \leftarrow\!- \lambda$$

This clause allows the goal statement <-- list(0,L) to have a refutation. The values that it computes for L are of the form: ω, $0.\omega$, $0.1.\omega$, $0.1.2.\omega$, etc...

The symbol ω, in this example, looks like the empty-list constant, and the values for L look like standard finite lists. Their pragmatics however is quite different, since the programmers can think in terms of infinite lists and not be worried about artificial terminal cases, which can be inserted systematically by the interpreter.
The introduction of the terminal clause is similar to the termination rule for infinite data productors proposed in [7]. In that case a process producing a (potentially) infinite data structure terminates when all the processes which consume that data structure have terminated (lazy evaluation). We obtain the same behaviour by exploiting the non-determinism of the language. A process which produces a (potentially) infinite stream, at each stream approximation can be reduced to λ. However, if there exist consuming processes, the process has an alternative reduction which produces a refinement of the stream.

The operational semantics is defined as follows:

If W is a set of clauses, and P is a predicate symbol of sort $s_1 \times \ldots \times s_n$, then the operational semantics of P in W is:

$$D_0(P,W)=\{(t_1,\ldots,t_n) \mid t_i \text{ has sort } s_i, i=1,\ldots,n \text{ and } P(t_1,\ldots,t_n) \mid \frac{\theta}{W'} \lambda\}$$

where W' is the union of the program W and of all of its terminal clauses, added accordingly to the rule above described.

EXAMPLE 1)

list(n,n.L) <-- list(s(n),L)

P(s(n),k.L,y) <-- P(n,L,m) , prod(k,m,y)
P(0,L,1) <-- λ

Assume <-- prod(k,m,y) be refutable iff y results to be the product of m and k.
list(n,L) is the process which produces the stream L of all the natural numbers starting from n.

P(n,L,m) defines the relation 'm is the product of the first n numbers in the stream L'.
Then, consider the program:

$$W' = \begin{cases} 1)\ fact(n,m) \leftarrow\!- list(1,L)\ ,\ P(n,L,m) \\ 2)\ P(s(n),k.L,y) \leftarrow\!- P(n,L,m)\ ,\ prod(k,m,y) \\ 3)\ P(0,L,1) \leftarrow\!- \lambda \\ 4)\ list(n,n.L) \leftarrow\!- list(s(n),L) \\ 5)\ list(n,\omega) \leftarrow\!- \lambda \qquad \text{(terminal clause)} \end{cases}$$

Note that 5 is the only terminal clause, since the clause $P(x,\omega,y) \leftarrow\!- \lambda$ will not satisfy our condition.

fact(n,m) defines the relation 'm is the factorial of n'.
We will now give an example of computation. For the sake of simplicity, the second clause will be rewritten in the form:

$$P(s(n),k,L,k*m) \leftarrow\!- P(n,L,m)$$

where the symbol '*' is interpreted as the product operator on natural numbers.

Initial goal statement:
$\leftarrow\!-$ fact(2,x)

by clause 1),and the substitution x=m:
$\leftarrow\!-$ list(1,L) , P(2,L,m)

by clause 2, and the substitution $L=k.L_1$, $m=k*m_1$:
$\leftarrow\!-$ list(1,k.L_1) , P(1,L_1 ,m_1)

by clause 2, and the substitution $L_1=k_1.L_2$, $m_1=k_1*m_2$:
$\leftarrow\!-$ list(1,k.k_1 .L_2) , P(0,L_2 ,m_2)

by clause 3, and the substitution $m_2=1$:
$\leftarrow\!-$ list(1,k.k_1 .L_2)

by clause 4, and the substitution k=1:
$\leftarrow\!-$ list(2,k_1 .L_2)

by clause 4, and the substitution $k_1=2$:
$\leftarrow\!-$ list(3,L_2)

by clause 5, and the substitution $L_2=\omega$:
$\leftarrow\!-\lambda$

The resulting substitution for x is:
$x=m=k*m_1=k*k_1*m_2=k*k_1=k_1=2$

The resulting substitution for L is:
$L=k.L_1=k.L_1.L_2=1.2.\omega$

Note that, to have a refutation, at least two elements of the list L have to be computed.

4. FIXED POINT SEMANTICS: FINITE APPROXIMATIONS

The fixed point semantics for a program W is defined as a model of the set of clauses W ∪ {terminal clauses}, obtained as the least fixed point of a transformation which is defined on the set of the interpretations of W [1,10,11].

The interpretations of W are defined over an abstract domain U, which is a family of sets U_s, each set being indexed by a sort s occurring in W.
Each U_s is defined as follows:

1) All the constant symbols of sort s, occurring in W, are in U_s (note that if s is a non-canonical sort, also ω_s is a constant symbol of sort s and then also ω_s belongs to U_s).
2) For each data constructor symbol of sort $s_1 \times \ldots \times s_n \rightarrow s$, U_s contains all the terms $d(t_1,\ldots,t_n)$ such that $t_1,\ldots,t_n$ belongs to $U_{s_1},\ldots,U_{s_n}$, respectively.

Note that U contains the standard many sorted Herbrand Universe as a proper subset, i.e. the set of all the ground terms in which none of the ω_s occurs. In addition U contains suspensions, i.e. non completely evaluated data, where both undefined and standard constant symbols occur. Finally, U contains also the fully undefined terms, i.e. the terms ω_s.

The Herbrand Base **B** of W is the set of all the ground atomic formulas: for each predicate P occurring in W, of sort $s_1 \times \ldots \times s_n$, and for each n-tuple of terms $t_1,\ldots,t_n$ belonging to $U_{s_1},\ldots,U_{s_n}$ respectively, $P(t_1,\ldots,t_n)$ belongs to **B**.

A Herbrand Interpretation I of W is any subset of **B** containing λ.

The set $\mathcal{J}$ of all the Herbrand Interpretations of W is partially ordered by the relation $\subseteq$ (set inclusion). As is the case for standard Horn clauses, $(\mathcal{J},\subseteq)$ is a complete lattice, i.e. for every possibly non finite subset $\mathcal{L}$ of $\mathcal{J}$, there exists lub($\mathcal{L}$) and glb($\mathcal{L}$).

It is possible to associate, to any program W, a transformation T on the domain of interpretations, defined in the following way:

$$T(I)=\{A \mid A \leftarrow B_1,\ldots,B_n \text{ is a ground instance of a clause of } W' \text{ and } B_1,\ldots,B_n \in I\} \cup \{\lambda\}$$

where W' is the union of the set W and of the terminal clauses for W.

It is well-known that the transformation T is monotonic and continuous [6].

Since T is monotonic, there exists:

$$I_F = \min\{I \mid I=T(I)\}$$

Moreover, since T is continuous:

$$I_F = \bigcup_{k\geq 0} T^k(\{\lambda\})$$

The fixed point semantics of a predicate P, of sort $s \times \ldots \times s_n$, in a program W is defined as follows:

$$D_F(P,W) = \{(t_1,\ldots,t_n) \mid t_1 \in U_{S_1}, \ldots, t_n \in U_{S_n}, P(t_1,\ldots,t_n) \in I\}$$

The equivalence of the operational and fixed-point semantics comes directly from the similar result for PROLOG.

5. FIXED-POINT SEMANTICS: INFINITE TERMS.

Now we want to define an alternative fixed-point semantics, which reflects the idea that non-canonical data, containing the symbols ω_S, are suspensions, that is partial approximations of infinite terms.

A term containing occurrences of the symbol ω_S cannot be transformed into an infinite term containing no occurrences of ω_S, because it would be necessary an infinite number of derivations. However it is possible to compare two suspensions to establish which is a better approximation.

Consider, for example, the process P(n,L) which produces the stream of all the odd numbers starting from n, if n is odd, and the stream of the even numbers starting from n, if n is even. Such process is defined by the clause:

1. P(n,n.L) <-- P(s(s(n)),L)

while the terminal clause is:

2. P(n,ω) <-- λ

One of the streams produced by the process P, starting from 0, is $L_1 = 0.2.\omega$, obtained by applying clause 1 twice and clause 2 once.
Another stream is $L_2 = 0.2.4.\omega$, obtained by applying clause 1 three times, and clause 2 once.
L_1 is a better approximation than L_2 of the stream which could be obtained starting from 0 and applying clause 1 forever:

$$0.2.4.6. \ldots$$

Clearly L_1 cannot be compared to any of the streams obtainable, for example, starting from 1 ($1.\omega$, $1.3.\omega$,

etc.).

It is then necessary to define a partial ordering $<$ on the elements of A (ground terms), which corresponds to the concept of "better approximation".

i) For any constant symbol c of sort s, $c_s < c_s$ and, if s is non-canonical, $\omega_s < c_s$.

ii) For any constructor symbol of sort $s_1 \times \ldots \times s_n \rightarrow s$:
   a) if $t_i = \omega_{s_i}$, $i=1,\ldots,m$, then $d(t_1,\ldots,t_n) = \omega_s$
   b) if $t_i < t_i'$, $i=1,\ldots,m$, then $d(t_1,\ldots,t_n) < d(t_1',\ldots,t_n')$

A similar partial ordering is defined on the Herbrand Base $\mathcal{B}$, as follows:

For any predicate symbol P of sort $s_1 \times \ldots \times s_m$, and for any $t_1,\ldots,t_m, t_1',\ldots,t_m'$ of sorts $s_1,\ldots,s_m$:
if $t_i < t_i'$ $i=1,\ldots,m$, then $P(t_1,\ldots,t_m) < P(t_1',\ldots,t_m')$.

Furthermore, it is necessary to introduce in the universe U all the infinite terms which are limits of monotonic sequences of terms. Similarly, it is necessary to introduce in the base $\mathcal{B}$ all the atomic formulas which contain infinite terms and which are limits of monotonic sequences of atomic formulas.

An interpretation of W is any subset of $\mathcal{B}$ which contains λ and which does not contain any pair formulas A and A', such that $A < A'$.

Obviously, the interpretation containing atomic formulas in which there occur infinite terms can be regarded as limits of monotonic sequences of interpretations without infinite terms.

Let ρ be a function which transforms subsets of B (containing λ) into interpretations. It is defined as follows: if s is a subset of B then

$$\rho(S) = S - \{A \mid A \in S, \exists A' \in S \quad A < A'\}$$

In other words ρ eliminates all those atomic formulas for which there exists in S a better approximation.

The set of the interpretations of W is partially ordered by the relation $<$ defined as follows: if I,J belongs to :

$$I < J \text{ iff } \forall A \in I \ \exists A' \in J \quad A < A'$$

or, equivalently:

$$I < J \text{ iff } I \in \sigma(J)$$

where σ is defined as follows:

$$\sigma(I) = \{A \mid \exists A' \in I \quad A<A'\} \cup \{\lambda\}$$

Note that, if I is an interpretation: $\rho(\sigma(I))=I$

The set $\mathcal{I}$ of the interpretations is a complete lattice with respect to $<$, and it holds, if $\mathcal{L}$ is a subset of $\mathcal{I}$:

$$\text{glb}(\mathcal{L}) = \rho(\cup\, \sigma(\mathcal{L}))$$
$$\text{lub}(\mathcal{L}) = \text{glb}(\mathcal{L}')$$

where $\mathcal{L}' = \{I' \mid \forall I \in \mathcal{L} \quad I<I'\}$

Note that $\mathcal{L}'$ is never empty, because it contains at least $\rho(\mathcal{B})$. In particular, if $\mathcal{L}$ is finite:

$$\text{lub}(\mathcal{L}) = \rho(\cup\, \sigma(\mathcal{L}))$$

The transformation T' associated to a program W is defined in the following way:

$$T'(I) = \rho(\{A \mid A \leftarrow B_1,\ldots,B_n \text{ is a ground instance of a clause of } W', \text{ and } B_1,\ldots,B_n \in \sigma(I)\} \cup \{\lambda\})$$

where W' is the union of W and of the terminal clauses of W.

$\sigma(I)$ occurs in the definition of I because, if a certain approximation of a data structure is computed, then also any less defined approximation of such a data structure must be considered as computed.

It can easily be proved that T' is monotonic and continuous, hence there exists the least fixed-point I'_F of T' and:

$$I'_F = \bigcup_{k\geq 0} T'^k(\{\lambda\})$$

The second fixed-point semantics is defined analogously to the first:

$$D_{F'}(P,W)=\{(t_1,\ldots,t_n) \mid t_1 \in U_{S_1},\ldots, t_n \in U_{S_n}, P(t_1,\ldots,t_n) \in \sigma(I'_F)\}$$

It is worth noting that in the previous semantics, the lub of the chain $T^k(\{\lambda\})$ contains only finite approximations (suspensions), while, for this semantics, the lub of $T'^k(\{\lambda\})$ can contain also infinite terms.

BIBLIOGRAPHY

1. Apt, K.R. and M.H. van Emden. "Contributions to the theory of logic programming". J. ACM 29 (1982).

2. Bellia,M., Dameri,E., Degano,P., Levi,G. and M.Martelli. "Applicative Communicating Processes in First Order Logic". Symposium on Programming. Lecture Notes in Computer Science 137 (Springer Verlag, 1982) 1-14.

3. Clark,K.L. and S.Gregory. "A relational language for parallel programming". Proc. of Functional Programming Languages and Computer Architecture Conf. (1981) 171-178.

4. Hannson,A., Haridi,S. and S.A.Tärnlund. "Properties of a Logic Programming Language". Logic Programming, Clark and Tarnlund Eds. (Academic Press, 1982) 267-280.

5. Kahn,G. and D.B.MacQueen. "Coroutines and networks of parallel processes". Information Processing 77, North Holland (1977), 993-998.

6. Kowalski,R. "Predicate logic as a programming language". Proc.IFIP Cong. 1974, North-Holland Pub. Co., Amsterdam, 1974, pp.569-574.

7. Monteiro,L. "An extension to Horn Clause Logic allowing the definition of concurrent processes". Proc.I.C.F.P.C. (Eds: J.Diaz, I.Ramos), LNCS 107, Springer-Verlag 1981.

8. Pereira,L.M. "A PROLOG demand-driven computation interpreter". Logic Programming Newsletter 4 (1982), 6-7.

9. Robinson,J.A. "A machine-oriented logic based on the resolution principle". J.ACM 12 (1965), pp.23-41.

10. Van Emden,M.H. and G.J. de Lucena. "Predicate logic as a language for parallel programming". Logic Programming, Clark and Tärnlund Eds. (Academic Press, 1982) 189-198.

11. Van Emden,M.H., Kowalski,R. "The semantics of predicate logic as a programming language". J.ACM vol.23 (1976) n.4, pp.733-742.

SOME ASPECTS OF THE STATIC SEMANTICS OF LOGIC PROGRAMS WITH MONADIC FUNCTIONS

Patrizia Asirelli

Ist. di Elab. dell'Informazione - C.N.R.
V. S. Maria, 46 - I56100 PISA - Italy

ABSTRACT

We consider logic programs in the Horn clauses form of logic with monadic functions, and present two approaches to derive a set of equations from a given set of clauses. The derivation is obtained by a data flow analysis of the variables, involved in each clausal definition. Each equation expresses the semantics of a procedure, by means of a set expression which, by transformation of the set of equations, can be reduced to a solved form. The set expression thus obtained represents, for each procedure its greatest, approximate, set of solutions; i.e. a set which contains the denotation defined, for the same procedure, by the standard semantics. The approximate solutions can be seen in the contest of abstract interpretations of programs, to get, by a static analysis of their definitions, some of their properties. The approximate solutions, being expressed by means of set expressions, could then be used as a tool for program verification and construction.

1. INTRODUCTION

We present an approach to the static analysis of programs, written in a simple logic language, defined as in [1-2], where procedures are defined by means of Horn clauses with monadic functions, and where all clauses are non negative. Thus a sort of very simple PROLOG [3-5].

We first define an algebraic semantics of clausal definitions, in the sense of representing possible set of solutions for a procedure, by means of equations and set expressions.

By static semantics of a program we mean all that can be deduced, statically, about the set of solutions for a logic program, as expressed by its standard semantics [2].

The aims of this paper fall into the same framework of [6], but for logic languages instead of algorithmic languages. As in [6], we get set expressions which represent in the most general

case, approximation to the set of denotations of a procedure as defined by the standard semantics. We also consider And/Or graphs [7-8], representation of programs instead of flow-charts. Thus, differently from [9], we do not try to get set expressions which denote the exact set of solutions, but only an approximation of it. At present the work is much semplified, with respect to [6] and [9], since we consider problems originated only by the use of monadic functions, treated symbolically, without looking, for now, for the fixpoints of their associated symbolic expression.

The same problem as been tackled for monadic logic programs with monadic functions; the next Section will give a brief summary of results obtained, for that case, in [10]. Section 3 will present two approaches to deduce, statically, a set of equations from a given set of clauses. In Section 4 we will present transfromations of such equations to get a set of equations in solved form. Section 5 contains few considerations on the defined transformations and their relations to other works. Section 6 extends the results of the previous sections, to clausal definition with monadic functions. We conclude with a brief summary in Section 7.
Appendix 1, at the end of the paper, gives an example of the construction of a set of equations for a given set of clauses, in the monadic case; Appendix 2 gives an example of a set of equations that can be obtained, according to the second approach presented in Section 3. In Appendix 3 a set of axioms is given to be used for transformations of equations obtained when functions are used in clausal definitions.

2. RESULTS FROM THE MONADIC CASE

Given a set of clauses A, defining n procedures Pi, monadic, we consider the set of clauses defining each procedure and its correspondent And/Or graph, [7-8]. Then we trace the values flow of the variable appearing at the root of the And/Or graph. The denotation of a procedure Pi, Dh(Pi), (results of the procedure Pi), can be derived in terms of union and intersection of denotations of the predicates involved in the definition of Pi. Thus we can derive, say, an $\cap/\cup$ graph, correspondent to the And/Or graph in object, by interpreting And nodes and Or nodes, respectively as intersection and union operations, and replacing each atomic formulas Qj(X) (where X is the variable traced), by the correspondent set Dh(Qj). By Dh(Pi) we denote the denotation of the predicate Pi, as defined by the operational semantics associated to Hyperresolution and Instantiation rules, [2]. Appendix 1 shows an example of the all process; from the $\cap/\cup$ graph there obtained, the following equation can be derived:

$$\{Pi\} == \left(\bigcap_{j=1}^{n} \{Qj\}\right) \cup \{f.\left(\bigcap_{j=1}^{m} \{Rj\}\right)\} \cup \left(\bigcap_{j=1}^{k} \underline{dom}(gj, \{Sj\})\right) \cup \{a\} \cup \{b\}$$

Where:
each {Rj}, {Qj} denotes the set of values, solutions of procedures Rj and Qj, respectively, derived from the static analysis of their clausal definition.
{f.T}, is a notation which, given a set (or set expression) T, denotes a set of data that, using a common set notation, can be expressed as: { f(y) | y ∈ T }.
dom(gj, {Rj}) denotes:- a set expression T such that:
{gj.T} ∈ {Rj};
- the empty set, {}, otherwise.
We have called 'Deduce' the function which produces a set of equation such as the above one, starting from a given set of clauses. Then we show, inductively, that when predicates are defined by clauses where: functions symbols are not used or else, they are used not recursively, then the set expression denoted by {P} can be computed to a set of values such that the following relation holds:

$$t \in Dh(P) \text{ iff } t \in \{P\} \quad \text{i.e. } Dh(P) = \{P\}$$

When clauses use function symbols recursively, either directly or not, that is, when clauses are such as follows:

P(f(X)) <- Q(X), P(X)

or

P(X) <- Q(X), R(X)
Q(f(X)) <- P(X)

then we Deduce equations of the form:

$$\{P\} == \{f.T\}$$

where T contains references to the symbolic set {P} itself. In those cases we need to find the fixpoint of such set expressions T. Then, if we are able to find a notation for such fixpoints so that by replacement of T in {f.T} we get a notation which is recursive, but self-contained and, if we are then able to define union and intersection between such sets then we can represent the denotation of a predicate by a set expression which is finite, independent of other predicates and which can be built by the static analysis of clauses. In [10] we suggest such a notation and give tranformation rules for deducing such notations from the set of equations obtained in the first place. On the other hand we show that those set expressions can be left symbolic, and transformed ({f.T}'s are only partially transformed), to obtain a set of equations, in solved form, which represent a new set of clauses. The new set of clauses are the semplified version of the set of clauses given in the first

place. Thus, we defined an algorithm Transf such that the following diagram commutes:

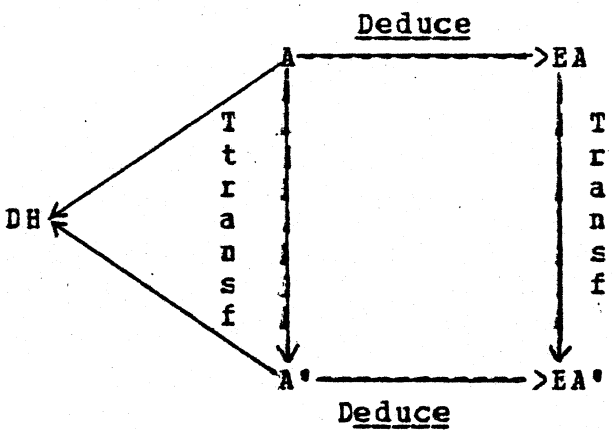


fig. 1

Where DH={Dh(P1),Dh(P2),....Dh(Pn)}, for all predicates P1,..Pn defined in A and A'.

That is, the set of equations EA' can also be derived from a set of clauses A', such that: denoting by $Dh_A(Pi)$, the denotation of Pi as defined by the standard semantics, when Pi is defined in a set of clauses A; then: A' and A are such that, for all procedures Pi defined in A, Pi is defined also in A' and:

$Dh_A(Pi) = Dh_{A'}(Pi) = Dh(Pi)$ if clauses in A do not contain local variables;

$Dh_A(Pi) \subseteq Dh_{A'}(Pi)$ otherwise.

Moreover, A' can be obtained by transformations of A.

Equations EA', obtained by transforming equations EA, may contain references to symbolic sets {Pi}, where Pi is an undefined predicate. Such symbolic sets can be eliminated (replaced by the empty set {}). In any case such equations either represent a ground set of values or else, they may be considered as patterns for the computation of them. They can also be used as a tool for programs development and programs composition, where symbolic set are used to define parametric specifications.

3. TWO APPROACHES TO THE EXTENTION OF DEDUCE

To introduce the problem of static semantics, for n-adic programs, let's consider the following clauses:

1) P(a,b) <-
2) P(a,X) <-
3) P(X,Y) <-Q(X,Y),P(X,Y)

4) P(X,Y) <- Q(X,Z) P(Z,Y)

In general we have clauses such as:

P(X1, X2,...Xn) <- Q1(t11,..tm1),...Qn(tn1,....tnk)

For the moment we do not consider functions, thus we assume all terms tij to be variables, either local or not.

We denote by {P} a set of tuples, each one of which is meant to represent a solution for P, according with the definition of Dh(P) in the same case.

Let's observe that Dh(P) in case of clause 1) is given by {<a,b>}; thus an obvious way to modify the Deduce function, is to produce the same results for clauses which are assertions. On the other hand, following the same approach, for the second clause we get a tuple such as {<a,?>}, where ?, means "all possible values", [10].
For the same clause it is:

Dh(P) = { <a, t> | ∀ t ∈ Herbrand Universe}

if we let {?} denote the Herbrand Universe of the set of clauses defining P, we can represent Dh(P) as: {a} X {?}. An obvious way to make things equal is then to define the cartesian products between sets as usual, so that:
{<a,?>} == {a} X {?}
as if ? were an other constant symbol. The semantics of {<a,?>} has to be defined as Dh(P) above, so that {P} and Dh(P) represent the same set of values. In general, a tuple as <a,b,?,c,d> represents a set of tuples whose first two and last two projections are fixed, and the middle one is one of all possible data. Given the meaning of such a notation, we can redefine Deduce so that, for all assertions, the following set expression will be constructed:

$\{< v_1, v_2, \ldots v_n >\}$ iff C : $P(t_1, t_2, \ldots, t_n)$ <-

for all $v_i = t_i$ if t_i is a constant symbol

v_i = '?' if t_i is a variable.

Let P be a m-adic procedure and let it be defined by n assertions; then we can deduce the following equation:

$\{P\} == \{ < v_1^1, v_2^1, \ldots v_m^1 >, < v_1^2, v_2^2, \ldots v_m^2 >, \ldots < v_1^n, v_2^n, \ldots v_m^n >\}$

If a procedure P has m arguments, than given the set of tuples {P}, we define the m projections of {P}, each one denoted by {P}_j, j=1:m. Thus, each {P}_j denote the set of values for the j-th argument of predicate P, and it is defined by:

$$\{P\}\_j == \{ v^i_j \mid \forall\, i=1:n \text{ and } \forall < v^i_1, \ldots v^i_j, \ldots v^i_m > \in \{P\} \}$$

Given {P} as above, it is, obviously:

$$\{P\} \subseteq \prod_{j=1}^{n} \{P\}\_j$$

The same holds when we Deduce each {P}_j, by the data flow analysis of variables, for all other type of clauses. We present two approaches: one leads to a set of equations which allows to find, for each given procedure Pi, an approximation of Dh(Pi); the other approach leads to a set of equations which could be refined to find an approximation of Dh(Pi), which is the closest one that can be found statically, by the data flow analysis of variables.

3.1 First approach

Like for the monadic case, we consider the And/Or graph which correspond to a clause; then we trace the value flows of all variables, arguments of the predicate being defined. For each variable Xi such that its trace binds it to the j-th argument of a call to procedure Q, we consider {Q}_j as the set originating values for that variable. As in the monadic case we then interpret as intersection all And nodes and as union all Or nodes. Just as an example, let us see that, proceeding as above, for clause 3 we would deduce:

$$\{P\}\_1 == \{Q\}\_1 \cap \{F\}\_1$$
$$\{P\}\_2 == \{Q\}\_2 \cap \{F\}\_2$$

While for clauses 4:

$$\{P\}\_1 == \{Q\}\_1$$
$$\{P\}\_2 == \{F\}\_2$$

We can see that {P}_1 X {P}_2 derived above, do not represent correctly the semantics of P as defined by the standard semantics for the corresponding clauses. In general, while for assertions, {P} and Dh(P), represent the same set of data, for all other clauses we have:

$$Dh(P) \subseteq \{P\}$$

That is, the static semantics, defined by the data flow analysis of variables, defines for a predicate P, a denotation which contains the denotation defined by the standard semantics. We can easily see that, for example, the standard semantics defines for P, relatively to clause 4, the following:

$$Dh(P) == \{ <t1,t2> \mid \forall <t1,k> \in Dh(Q) \text{ and } <k,t2> \in Dh(F) \}$$

while the set expression we get for {P}, can be expressed as:

$$\{P\} == \{\langle t_1, t_2\rangle \mid \forall\ k_1, k_2 : \langle t_1, k_1\rangle \in \{Q\} \ \underline{\text{and}}\ \langle k_2, t_2\rangle \in \{F\}\}$$

To conclude, given a set of clauses A, for each predicate Pi, defined in A, by this approach we would get a set of equations as follows:

$$\{Pi\} == \bigcup_{j=1}^{n} \prod_{k=1}^{m} \{Pi\}j\text{-}k$$

for Pi defined by n clauses, and Pi being a m-adic procedure.

$\{Pi\}j\text{-}k == T$ for T a set expression defining the set of possible values for the k-th argument of Pi, defined by the j-th clause.

Let's observe that this way we find the greatest approximation of each Dh(Pi) that can be found by this method. Let's also observe that the set of equations EA are such that the following diagram commutes:

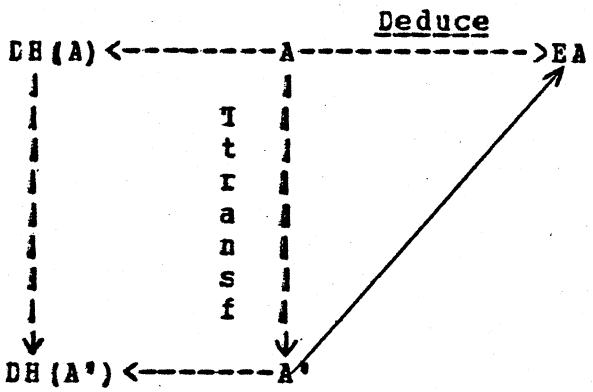


fig. 2

Where $DH(A) = \{Dh_A(P1), Dh_A(P2), \ldots Dh_A(Pn)\}$

$DH(A') = \{Dh_{A'}(P1), Dh_{A'}(P2), \ldots Dh_{A'}(Pn)\}$, and $Dh_A(Pi)$, $Dh_{A'}(Pi)$ defined in Section 2, fig. 1.

In fact we can easily prove that, given a clause with local variables that binds together some procedure calls, ignoring local variables completely, we get an approximation of its denotation, which is the denotation of an analogous clause, where all local variables, in all procedure calls are different, one from each other. That is, given, for example:

P(X,Y) <- Q(X,Z1), F(Z1,Y), R(Y,Z2), M(Z2,Z1)

Indeed we find the denotation of P defined as:

P(X,Y) <- Q(X,Z1), F(Z2,Y), B(Y,Z3), M(Z4,Z5)

Thus we get a set of equations EA, that can also be derived from a set of clauses A', such that the denotation of all procedures, defined in A are contained in the denotation of the same procedures defined in A'. I.e.
$Dh_A(Pi) \subseteq Dh_{A'}(Pi)$ for all procedures Pi.
The previous relation shows that by ignoring local variables, the method of data flow analysis of variables, ensure partial consistency with standard semantics. In fact, the method allows to find sets of values such that, some of the values are correct solutions, while others aren't. Yet no value, outside those sets, can be a correct solution.

3.2 Second approach

To obtain a set expression for P representing a set, nearer to Dh(P), we should define, given clause 4 above, two subsets for {Q} and {F}, $\{\underline{Q}\}$ and $\{\underline{F}\}$ respectively, such that:

$\{\underline{Q}\} == \{\langle t_1, t_2\rangle \mid \exists t_3 : \langle t_1, t_2\rangle \in \{Q\} \text{ and } \langle t_2, t_3\rangle \in \{F\}\}$

$\{\underline{F}\} == \{\langle t_1, t_2\rangle \mid \exists t_3 : \langle t_1, t_2\rangle \in \{F\} \text{ and } \langle t_3, t_1\rangle \in \{Q\}\}$

The previous sets can also be obtained as follows:

$\{\underline{Q}\} == \{Q\} \cap (\{Q\}_1 \times (\{Q\}_2 \cap \{F\}_1))$

$\{\underline{F}\} == \{F\} \cap (\{F\}_1 \cap \{Q\}_2) \times \{F\}_2))$

Then, defining $\{P\}_1$ and $\{P\}_2$ we should consider $\{\underline{Q}\}_1$ and $\{\underline{F}\}_2$ instead of {Q} and {F}. Yet, although $\{\underline{Q}\}$ and $\{\underline{F}\}$ represent the set of tuples of Q and F, satisfying clause 4, because of the cartesian product X, {P} would still be greater than Dh(P); i.e.: denoting by $\{\underline{P}\}$ the set obtained by considering $\{\underline{Q}\}$ and $\{\underline{F}\}$ instead of {Q} and {F}, it is:

$Dh(P) \subseteq \{\underline{P}\} \subseteq \{P\}$

Thus $\{\underline{P}\}$ would still be an approximation of Dh(P).
Everything previously said about partial consistency, still holds. Yet $\{\underline{P}\}$ is less approximate than {P}.

The set expression $\{\underline{P}\}$, represents the least approximation we can get for the set of solutions of P, by the static analysis of its clausal definiton, following the approach of tracing variables. This happens just because the clausal definition of P was such that only two procedure calls shared a variable. In general if n procedure calls are such that each one shares one

(or more), variable with others, then the least approximate solution need to be found by an iterative process:

1) First define each {Qi} for each procedure call; this would give a restriction of {Qi} in terms of other procedures with which Qi shares its arguments. Since the same is done for all procedure calls, there may be tuples of other procedures, satisfying the sharing conditions with Qi, which do not satisfy other sharing conditions in the same clause. This, in general, means that:
2) We need to define a {Qi}' identical to {Qi} but where the sets involved are the restricted ones, {F}'s instead of {F}'s.
3) The process of refining {Qi} has to go on until we get two sets, say {Qi}' and {Qi}", such that, either {Qi}'={Qi}", or else {Qi}" is empty.

This, informally described, refining process, will terminate. Let's in fact remind that, for all predicates Q and F, with two arguments (the same holds for predicate with any number of arguments), it is:

$\{Q\} \subseteq \{Q\}_1 \times \{Q\}_2$, and also:

$\{Q\} = \{Q\} \cap (\{Q\}_1 \times \{Q\}_2)$ and

$(\{Q\}_1 \cap \{F\}_1) \times (\{Q\}_2 \cap \{F\}_2) \subseteq \{Q\}_1 \times \{Q\}_2$ and

$(\{Q\}_1 \cap \{F\}_1) \times (\{Q\}_2 \cap \{F\}_2) \cap \{Q\} \subseteq (\{Q\}_1 \times \{Q\}_2) \cap \{Q\} \subseteq \{Q\}$

Thus, the refining function is monotone descendent and it stops, either producing an empty set, {}, or producing the same set. Moreover, when the process stops, all sets such as {Q} and {F}, represent, the exact set of tuples satisfying all conditions in the clause. Than the set of solutions for the procedure defined by the clause in object, should be build in terms of these last sets.

The above process can be defined, perhaps more clearly, if we consider all variables, either locals or not, involved in a clause. For each variable we define a set expression representing its possible values, deducing such set expression from the And/Or graph of the clause. Thus, given a clause such as:

P(X1, X2,...Xn) <- Q1(t11,..tm1),..., Qn(tn1,....tnk)

Let's denote by X the set X1,X2,..,Xn and by Y the set Y1,Y2,...Yw of local variables in the previuous clause. Then for some of the above tij it is either

tij $\subseteq X$, or Tij $\subseteq Y$.

Consider the And/Or graph, G, correspondent tc the above clause, and trace all variables in it. Then for each variable Zi, with Zi $\in X$, or Zi $\in Y$, consider the subgraph GZi of G which correspond to the trace of Zi, and trasfcrm it as follows:

-the rcot is labelled by {σ1Zi}

-all And nodes become nodes; all Or nodes become U nodes;

-all nodes Qj(tj1,....tjm), are replaced by an node with k arcs leading out, respectively, to a nodelabelled by {Qj}_k; for Qj(tj1,....tjm) such that there exists a k= s:r, with 1≤s≤r and tjk=Zi. Appendix 2 gives an example.

We can deduce {σ1Qj}, in the same way we deduced {Qj}, by defining such {σ1Qj} in terms of {Qj} and {σ1Zi}, depending on the variables of Qj in the clause. Making sure that by i we always get a different set identifier, we can get a set of equations which can be transformed into sone solved form, by a transformation process analogous to the one presented in next section. The set of equations in solved form, thus obtained, can be transformed again by the abcve mentioned refining functicn.

The set of equations we get by this approach can be summarized as follows: Given a set of clauses A,
- let P be the set of procedure symbols, defined, or just used, in clausal definitions of A;
- let {P1,P2,....Pn} $\in P$, be the set of procedure defined in A;
then, for all Pi an equation is built which locks as follows:

$$\{Pi\} == \bigcup_{j=1}^{u} \left(\prod_{k=1}^{m} \{\sigma j X_{jk}\} \right) \quad \text{with } X_{jk} \in X_{ij} \text{ and } X_{ij} \text{ the set}$$

of variables, terms of Pi in the j-th clause.

Let Z_{ij} be the set of all variable symbcls, local or nct, in the j-th clause defineg Pi, then: For each variable symbol Yjk $\in Z_{ij}$, we have a set of equations as follows:

{σjY$_{jk}$}== T where T is a set expression containing symbolic sets such as {σjQw}.

For each procedure Qw, called in the j-th clause defining Pi, we have a set of equations such as:

$$\{\sigma j Qw\} == \{Qw\} \cap \left(\prod_{s=1}^{r} \{\sigma j Y_{js}\} \right)$$ if r-1 is the number of argument of Qw and all Y_{js} are the variables, terms of Qw, in the j-th clause defining Pi.

4. TRANSFORMATION OF EQUATIONS

We will only consider the approach seen in 3.1, since we believe that the transformations we are going to define, can be accordingly modified to be applied to equations as defined in 3.2 above.

4.1 The transformation process

Thus, from 3.1 above, we have that: given a set of clauses A, Deduce(A), produces a set of equations, as in the monadic case, with the further complications of projections. As to transform the set of equations define in 3.1 above, let us observe that the best result we would like to get, is a a set of equations, each one of which associates a ground set (a set of constant symbols), to a procedure. Since some procedures may be defined in terms of undefined procedures, and since we believe that this is a useful information to keep, we want final set expressions, associated to procedures to maintain such references. Thus:

We say that a set of equations is in 'solved form' if and only if, each equation has the form:

{Pi}==T or {Pi}j_k== T

for all procedure symbols Pi, and all integer j and k, and all set expression T such that:

1) T is a ground set; else
2) T is the empty set {}; else
3) T is a set expression which contains symbolic sets {Qj} and such that {Qj} does not appear on the left-hand side of any other equation (Qj is an undefined procedure).

We define now the following transformation algorithm

Transf1:
1) apply rule RR1;
2) apply RR, until possible;
3) apply S axioms, until possible;

Replacement Rule 1 (RR1)

For all equations of the form: $\{Pi\} == \bigcup_{j=1}^{n} \prod_{k=1}^{m} \{Pi\}j\_k$

replaces each occurence of '{Pi}_k', in all set expressions of all other equations, by

$\bigcup_{j=1}^{n} \{E\}j\_k$

Since from Deduce we get equations such as, for example:

{P}== ({P}1_1X .. X{P}1_m) U .....U ({P}u_1X .. X{P}u_m)

{P}1_1== ({M}_1 ∩ {T}_2) X ({E}_1 ∩ {D}_3)
...
{P}1_m==.....
...
...
{P}u_1==....
...
{P}u_m==.....

{M}== ({M}1_1 X.. X{M}1_mm) U ...U ({M}k-1 X.. X {M}k_mm)
...
Rule BR1 allows to eliminate all references of type '{P}_w' and replaces them by a more detailed set expression in terms of '{P}j_w''s. By BR1, the previous example would be transformed in:
{P}== ({P}1_1 X .. X{P}1_m) U .....U ({E}u_1X.. X{P}u_m)

{P}1_1== ({M}_1 ∩ {T}_2) X (({P}1_1 U..U {P}u_1) ∩ {D}_3)
.....
with all other equations modified accordingly.

Elimination Rule (ER)

Given an equation such as: {Pi}j_k== T
such that T contains ' {Pi}j_k', replaces '{Pi}j_k' in T, by the empty set {}.

Synthesis axioms (S)

-For all set expressions T:
{} U T == T
{} ∩ T == {}
-For all ground sets D_i, i=1,n : $\prod_{i=1}^{n} D_i$ is defined as usual;
Operations of U and ∩ are defined for ground sets as usual; moreover S will contain axioms for associativity and distributvity of U and ∩ .

Rule EB, defined for the monadic case, as been modified into rule ER above, to take into account projections of tuples. For the monadic case rule ER was an obvious consequence of the observation that, given a recursive clause such as:

P(X) <- Q(X),R(X),P(X)

according to its standard semantics, no denotations, different from those generated by all other clauses defining P, will be generated by that clause. In the n-adic case, the analogous

happens for recursion over the same argument of a procedure. That is, consider the following clause:

P(X,Y)<-P(X,Z),B(Z,Y)

From the standard semantics we have that the previous clause adds tuples to the denotation of P, defined by the other clauses defining P, in the sense that it adds tuples where only the second elements may be new. Said it another way, considering the greatest set of solutions for P, denoted by: Dh(P)-1 X Dh(P)-2, the above clause may add elements to Dh(P)-2, not to Dh(P)-1. Thus, since we are now looking for the greatest approximation of Dh(P), we can consider the above clause having the same greatest set of solutions of P defined as:

P(X,Y)<-P'(X,Z),B(Z,Y)

P(X,Y)<-P'(X,Y)

and where P' is defined as the rest of P. That is, if P was defined by u clauses and the one considered previously was the k-th, then P' is defined so that:

$$Dh(P') == \bigcup_{i=1}^{k-1} Dh(P)i \cup \left(\bigcup_{j=k+1}^{u} Dh(P)j \right.$$

Thus we define RR so that each time we have equations such as:

{P}i_j= ({P}i_j ∪ I1) X (...

we replaces all occurences of '{P}i_j', on the left hand side of the previous equation, by the empty set '{}'.

Transf1 as defined above will certainly stops, since the number of substitutions RR1 has to do is finite; moreover, each substitution produces a set of equations such that the same type of sets {P}_k, for all predicates, will not appear anymore in anyone clause, unless P is undefined (thus no equation exists for {P}); RR1 needs to be done only once.
The same, of course, holds for RR;
S axioms are obviously convergent, and a point will be reached so that no one of them can be applied anymore.

Further transfromations are defined by the following algorithm:

Transf2

1) apply RR2;
2) apply RR;
3) apply S axioms;
4) repeat from 1) to 3) until all of them cannot be applied anymore.

Replacement Rule 2 (RR2)

Given an equations of type: {P}j_k= T
where T is a set expression; replaces each occurence of '{P}j_k' by T in all set expressions of all other equations.

EE and S axioms are define as in Transf1.

Rule RR2 is a transformation analogous to RR1.

Transf2 stops, as well as Transf1 does. Let's in fact observe that:
RR2 is applied after transf1 is completed; no equation, such as {P}j_K=T, will be such that T contains '{P}j_k' itself (because of EE in Transf1);
At each step, RR2 eliminates all references to sets such as '{P}j_k'; thus, the next time RR2 won't be applied to the same equation; since this applies to all equations and since there are a finite number of equations, after a while, RR2 will not be applicable anymore.
Because of the same sort of considerations, about EE and S axioms, we can conclude that Transf2 terminates, producing a set of equations each one of which has associated: either a ground set, or {} or else a set expression containing references to undefined procedures, i.e. it produces a set of equations in solved form.

At the end, since references to {P}j_k do not appear in any set expression and since they where built just for the sake of transformations, all equations for such sets can be eliminated; all that will remain is a set of equations, for the procedures defined in the set of clauses given in the first place.

5. FEW REMARKS ON TRANSFORMATIONS

The transformation process, presented in Section 4, is such that the following diagram commutes:

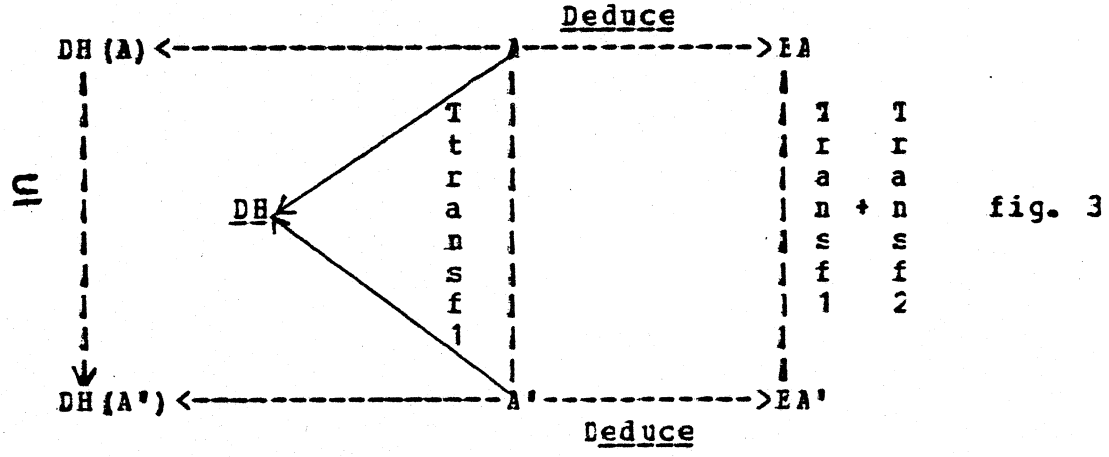


fig. 3

In fact, Transf1 and Transf2 steps, producing a set of equations, EA', in solved form; thus, the algorithm given by Transf1 followed by Transf2 is complete. It ensures a solution to the set of equations given in the first place.

Furthermore, the set of equations EA', can be deduced from a set of clauses A', such that A and A' have the same greatest approximate set of solutions, DH, with: DH={Dh(P1),....Dh(Pn)}, where each Dh(Pi) is the greatest set of approximate solutions of Pi, for all procedures Pi defined in A.
In fact, BR1, BR2 and S axioms, do not alter the semantics of the set of equations they are applied to; thus, the set of clauses correspondent to the set of equations, before and after BR1, BR2 and S axioms, can be obtained by a similar transformation of clauses in A.

The transformation process given in Sec. 4, is equivalent to the non-deterministic algorithm, given in |11|, which transform a given set of equations into another one, in soved form, to find an efficient unification algorithm. BR1 and BR2 are analogous to 'Variable Elimination', |11-12|, and to the Unfolding transfromation defined in Program Transformations, |13-14|. ER is analogous to the transformation which erases equations such as x=x, in |11|, and Compaction in |12|. Also ER is equivalent to represent by {} the failure of transformations, |11| for equations: x=t, where t contain x.

For all procedures Pi, defined in A and A', their denotations, as defined by the standard semantics is such that:

$Dh_A(Pi) \subseteq Dh_{A'}(Pi)$ thus $DH(A) \subseteq DH(A')$

with DH(A), DH(A'), $Dh_A(Pi)$ and $Dh_{A'}(Pi)$ defined as for fig. 1, 2.
The above results is due to the method of not considering local variables at all, as it has been shown in Section 3.1.

6. CLAUSAL DEFINITIONS WITH MONADIC FUNCTIONS

In Section 2 we introduced a notation for functions, used in |10|. Now we are going to see how to extend results of previous sections 3-4, for clauses with functions.

Functions can be, either in terms of procedure being defined by a clause or else, be terms of procedure calls.
Let's first observe that:

P(f(X)) <- Q1(X),....Qn(X) is equivalent to:

P(f(X) <- P'(X)
P'(X) <- Q1(X),....,Qn(X)

Therefcre, we can consider {P'} to be egmivalent to {P} where the clausal definition of P does nct ccntain any function, on its definition part. I. e.

$$\{P\} == \{f.\{P'\}\} \text{ and } \{P'\} == \bigcap_{i=1}^{n} \{Qi\}$$

Ihus fcr the above definition of P, we can deduce:

$$\{P\} == \{f.(\bigcap_{i=1}^{n} \{Qi\})\}$$

Ihus, when functions are terms of a procedure being defined by a clause, the result of the procedure, relatively to that particular argument, in that particular clause, is given by:

$\{P\}\_j\_k == \{f.T\}$

where T is derived in the same way as in 3.1, as if the function f didn't appear at all.

On the other hand if a variable X, argument cf a procedure definition, is also the argument of a functicn 'g', in a procedure call to Q, then, instead of considering $\{Q\}\_k$, we will consider: dom(g, $\{Q\}\_k$). Which is a consequence cf the meaning of dcm (Section 2), and the consideration that:

P(X) <- Q(f(X)

is such that the solutions for P, will be all those values 'v', such that: f(v) ∈ Dh{Q}, i.e. a set D such that '{f.D}' ∈ {Q}. For example, from:

P(f(X),g(X),m(n(Z))) <- Q(X,h(Y)), R(m(X),Z) we have :

$\{P\}\ 1\_1 == \{f.(\{Q\}\_1 \cap dom(m, \{R\}\_1))\}$
$\{P\}\ 1\_2 == \{g.dom(h, \{Q\}\_2)\}$
$\{P\}\ 1\_3 == \{m.\{n.\{R\}\_2\}\}$

The second approach in 3.2, needs to be slightly modified according to the previous notaticns.

Fcr what concern transformations, rule BR1 and RR2 need to be modified so that replacements are not applied to references in non-atomic sets (i.e. sets such as {f.T}) and in arguments of the dcm function. The solved form of set expressions is thus, such that symbolic sets, defined by cther equaticns may appear only in set expressions which are part of non-atcmic sets, or argument of the function dom. Everything previously said about transformations in Section 4, still holds.

At this point a further transformation can be done to expand a bit more set expressions in non-atomic sets, and in arguments of dom, to get some more information about the results of procedures, avoiding non termination of transformations, because of recursive set expressions. Although it will not be dealt with in this paper, we believe a notation, for non-atomic sets, can be found, such that, by a similar process of transformations, the fixpoint of such set expressions can be derived. We will then be able to represent data, built by recursive applications of functions, by a self-contained, recursive symbolic expression. For the moment we propose the following further transformations, for the set of equations obtained by the modified algorithms Transf1 and Transf2.

Transf3:

- apply RR1 so that replacents take places in non-atomic sets and in set expressions, argument of dom.
- given a set of n equations, choose one of the form: {P}j_k==I;
1) replace each occurense of the left hand side of the given equation, by its right hand side, in all set expressions of other equations.
2) apply SS axioms;
3) choose an equation of the form {P}j_k==I, which has not been choosen yet;
4) repeat 1-3, until all equations have been chosen once.

Axioms SS (old S axioms plus axioms for non-atomic sets and dom expressions) are listed in Appendix 3. They can be proved convergent and consistent with the meaning of non-atomic sets and the dom function. The set expressions still represent approximate solutions of procedures.

7. CONCLUSIONS

We have considered logic programs in the Horn clauses form of logic with monadic functions. We have then presented two approaches to derive, from a given set of clauses a set of equations. The set of equations obtained represent, for each procedure, its greatest, approximate, set of solutions; i.e. a set which contains the denotation defined by the standard semantics.

Equations are derived from clauses by a data flow analysis, for the variables involved in the clause, carried on over the correspondent And/Or graph.

Given a set of equations (derived as in the first one of the two approaches presented), we define a transformation algorithm which reduces equations to a solved form. The set of equations thus obtained is such that each equation expresses, by a set expression, the set of approximate solutions for a given

procedure. By approximate set of solutions for a procedures, we mean a set of values (when possible) some of which are correct solutions for the given procedures, while some others are wrong solutions. In any case, no other values, outside the approximate set of solutions, can be correct.

The aim of the paper is not to find tranformations in order to obtain more efficient programs, as it is the case for Program Transformations and Synthesis [13-14-15]. Our aim instead, is to find some properties of a program, from the static analysis of its definition, in the framework of Abstract Interpretations of Program, [6]. It is because of this that, for example, we believe that refences to undefined procedures should be kept in set expressions, for they could be used as a tool for program verification, program construction and composition of programs which have been defined separately.

REFERENCES

[1] Kowalski, R.A. - Predicate Logic as Programming Language, Proc. Information Processing 74, North Holland Pub. Co., Amsterdam, pp.569-574, 1974.

[2] Van Emden, M. H. and Kowalski R.A.- The Semantics of Predicate Logic as a Programming Language, Journal ACM, Vol. 23, No. 4, pp. 733-742, Oct. 1976.

[3] Colmerauer, A. et al. - Un Systeme de Communication Homme-Machine en Francais - Rapport preliminaire, Groupe de Researche en Int. Art., Universite d'Aix - Marseille - Luminy, 1972.

[4] Warren,D., Pereira, L., Pereira, F. - PROLOG: the language and its implementation compared to LISP, Proc. Symp. on A.I. and Prog. Lang., SIGPLAN Notices, Vol. 12, No. 8, and SIGART News. No. 64, pp. 109-115, Aug. 1977.

[5] Clark,K. L. and McCabe, F. - IC-PROLOG reference Manual, CCD Research Report, Imperial College, London, 1979.

[6] Cousot,P., Cousot, R.- Abstract Interpretation : A Unified Lattice Analysis of Programs by Construction or Approximation of Fixpoints. In : SIGACT/SIGPLAN Conf. Rec. of the Fourth ACM Symp. on Principle of Programming Languages; Los Angeles, Cal., jan. 17-19, pp. 238-252, 1977

[7] Kowalski, R.A. - Logic for Problem Solving; Artificial Intelligence series, (Nilsson, N.L. ed.), North Holland, 1979.

[8] Levi, G., Sirovich, F.- Generalized And/Or Graphs. Artificial Intelligence, 7, pp. 243-259, 1976

[9] Marque-Pucheu, G.- Equations booleennes generalisees et Semantique des programmes en logique du premier ordre monadiques. PhD Thesis, Ecole Normale Superieure, Paris.

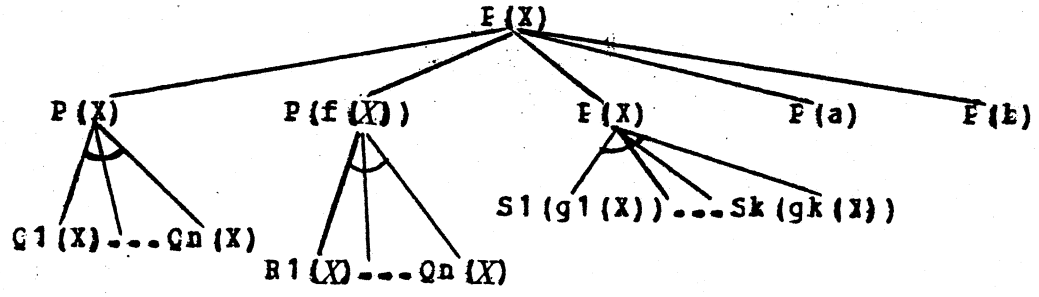
[10] Asirelli, P. - Horn Clauses Form of Logic: Algebraic Static Semantics of Programs. Int. Rep. I.E.I., B82-23, 1982.

[11] Martelli, A., Montanari, U.- An Efficient Unification Algorithm, ACM Trans. on Prog. Lang. and Systems, Vol. 4, n. 2, April 1982.
[12] Colmerauer, A.- PROLOG and Infinite Trees, in Logic Programming, K.L. Clark and S.-A. Tärnlund eds., Accademic Press, 1982.
[13] Burstall, R.M., Darlington, J.- A transformation system for developing recursive programs. Journal of the ACM 24, No. 1, 46-67, 1977.
[14] Clark, K.L., Darlington, J.- Algorithm classification through synthesis. The Computer Journal 23, No. 1, 1980.
[15] Burstall, R.M. : Recursive programs: Proof, transformation and synthesis. In "Rivista di Informatica" 7, pp. 25-42, 1976.

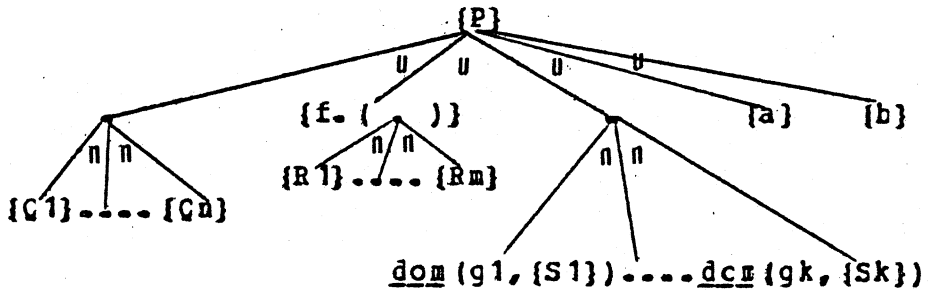
APPENDIX 1

P(X) <-Q1(X),....Qn(X)
P(f(X)) <-R1(X),....Rm(X)
P(X) <-S1(g1(X)),...Sk(gk(X))
P(a) <- P(b) <-

Its correspondent And/Or graph can be drawn as follows:

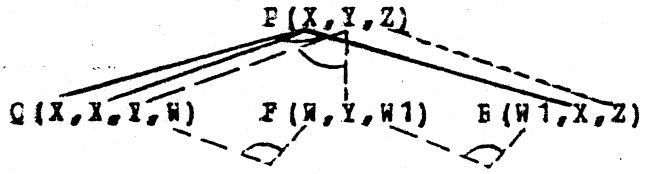


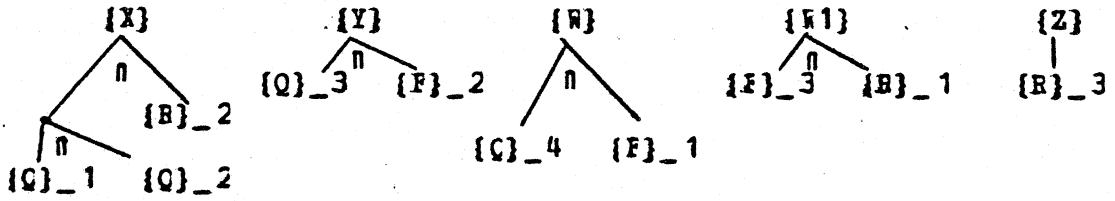
From the previous graph, we can deduce the following:



APPENDIX 2

P(X,Y,Z) <- Q(X,X,Y,W), F(W,Y,W1), B(W1,X,Z)





{P}== {X} X {Y} X {Z}

{X}== ({Q}_1 ∩ {Q}_2) ∩ {B}_2
{Y}== {Q}_3 ∩ {F}_2
{Z}== {R}_3
{W}== {Q}_4 ∩ {F}_1
{W1}== {F}_3 ∩ {R}_1

{Q}== {Q} ∩ ({X} X {X} X {Y} X {W})
{F}== {Q} ∩ ({W} X {Y} X {W1})
{B}== {B} ∩ ({W1} X {X} X {Z})

APPENDIX 3

1- For all set expressions A and B, such that A and B are ground sets:

- A ∪ B == { x | x ∈ A or X ∈ B}
- A ∩ B== { X | x ∈ A and x ∈ B}

2- For all set expression D :

- D ∪ {} == D
- D ∩ {} == {}
- D ∪ {?} == {?}
- D ∩ {?} == D
- (D ∪ B) ∩ D = D ∩ (D ∪ B) =D ∩ (B ∪ D) == D
- (D ∩ B) ∪ D = D ∪ (D ∩ B) =D ∪ (B ∩ D) == D

3) For all atomic sets D, and all non-atomic sets H

- $D \cap H == \{\}$

4) For all non-atomic sets, and for all l, w and n:

- $\{f_l . \{f_w ... \{f_n . \{\}\}..\}\} == \{\}$

- $\{f_l . D\} \cup \{f_l . \{?\}\} == \{f_l . \{?\}\}$

- $\{f_l . D\} \cap \{f_l . \{?\}\} == \{f_l . D\}$

5) For all non-atomic sets and all set expressions H, D and all l, k, s:

- $\{f_l . H\} \cap \{f_k . D\} == \{\}$ iff $l \neq k$

- $\{f_l . \{f_k . H\}\} \cup \{f_l . \{f_s . D\}\} == \{f_l . (\{f_k . H\} \cup \{f_s . D\})\}$

- $\{f_l . \{f_k . H\}\} \cap \{f_l . \{f_s . D\}\} == \{f_l . (\{f_k . H\} \cap \{f_s . D\})\}$

-For all functions f:

$\underline{dom}(g, \{\}) == \{\}$

- and for all ground sets D:

$\underline{dom}(g, D) == \{\}$

-For all set expressions T:

$\underline{dom}(g, T) \cup \underline{dom}(g, T) == \underline{dom}(g, T)$

$\underline{dom}(g, T) \cap \underline{dom}(g, T) == \underline{dom}(g, T)$

$\underline{dom}(g, \{g.T\}) == T$

$$\underline{dom}(g, \bigcup_{i=1}^{n} T_i) == \bigcup_{i=1}^{n} \underline{dom}(g, T_i)$$

$$\underline{dom}(g, \bigcap_{i=1}^{n} T_i) == \bigcap_{i=1}^{n} \underline{dom}(g, T_i)$$

A FIRST ORDER SEMANTICS OF A CONNECTIVE SUITABLE TO EXPRESS CONCURRENCY

Pierpaolo Degano Stefano Diomedi

Dipartimento di Informatica

Universita' degli Studi di Pisa

Abstract. The paper presents an extension to PROLOG that allows to directly express concurrency and synchronization. This is achieved by introducing the concept of class, a sort of cluster made of concurrent atoms. In general, a set of clauses involving classes is equivalent to a denumerable infinite set of pure PROLOG clauses. First, syntax and operational semantics of our extension are defined. Then a first order semantics is given that slightly generalizes classical PROLOG model-theoretic semantics; a fixpoint semantics is also given. Finally, an example illustrate the expressive power of the extension.

1. INTRODUCTION

Recent achievements in hardware technology made it feasible the development of machines that can directly execute logic programming languages. Among these, PROLOG is the most relevant both for theoretical and for practical reasons [2,6]. However, PROLOG is not satisfactory enough to conveniently express the concurrent features that hardware provides nowadays. As a matter of fact, PROLOG procedures can be naturally executed either in a parallel or in a co-routining fashion. The former regimen is simply achieved by simultaneously replacing a set of independent atoms in the current goal. Co-routining occurs when the same variables are shared by different atoms, thus realizing a sort of asynchronous communication. Unfortunately, there is no explicit way of synchronizing the computations of two or more concurrent processes, as is required when they cooperate to solve the same problem.

In order to solve this limitation, a number of extensions to PROLOG have been introduced [3,4,7,9]. All these extensions allow to write clauses with more than one atom in their left-hand side, e.g.

A(x,y) & B(y,z) <-- C(x,y,z), D(z,w)

where variable y acts as a synchronous communication channel between atoms A and B. The intended operational meaning of such a clause is that suitable instances of atoms C and D can be replaced for an instance of A and B, only when both of them are present at the same time in the goal.

The aim of this paper is to give a formalization of the above operational meaning within a logic framework, so that all the appealing semantic features of PROLOG carry over this extension. Moreover, we claim that the notions of synchronization and communication will be better understood and expressed by precisely stating the meaning of clauses such as the one above.

First, the paper describes the syntax of both the left- and right-hand sides of clauses along with the language operational semantics; then it defines a first order semantics which is a straightforward generalization of the one given by vanEmden and Kowalski [5]. A fixpoint semantics is also given, and the three different semantics are shown to be equivalent. Finally, the paper shows how a concurrent program can be translated in a pure PROLOG program, generally composed by a denumerable set of clauses.

2. SYNTAX AND OPERATIONAL SEMANTICS

In this section we will give the syntax of our extension to PROLOG in two steps. First, we will introduce concrete syntax. It is an abbreviation for some constructs of the abstract syntax that will be defined later.

The concrete syntax of the language is the following.

A program is a set of clauses.

A clause is a sentence of the form

X <-- B1 + ... + Bm

where X is a class and each Bi is an atom.
The formula B1 + ... + Bm is the (possibly empty) body of the clause and X is its header.

A class either is an atom or has the form

(A&X)

where A is an atom and X is a class.
The notation (X&A) is completely equivalent to (A&X).

An atom has the form

$$A(t1,\ldots,tn)$$

where A is a predicate symbol and each ti is a term, i=1,...,n.

A term is built by variables and constructor applications to terms.

A goal is of the form

$$<\!-\!-\ B1 + \ldots + Bm \qquad m\geq 0.$$

The concrete syntax allows to abbreviate goals and bodies by using the connective +. Let us now define abstract syntax that gives to + a meaning in terms both of standard first order logic connectives, and of classes.

The formula

$$A1 + \ldots + An$$

is an abbreviation for

$$\begin{array}{l}(A1 \wedge \ldots \wedge An) \vee \\ (X11 \wedge \ldots \wedge X1k_1) \vee \\ \ldots\ldots \\ (Xp1 \wedge \ldots \wedge Xpk_p) \vee \\ (A1\& \ldots \& An)\end{array}$$

where:
- each Xij is a class built with atoms Ak;
- each Ak belongs exactly to one class Xij;
- p+2 is the number of all the possible conjunctions of distinct classes obtainable from A1, ... ,An. Actually,

$$p+2=\sum_{k=1}^{n} s(n,k)$$

  s(n,k) being the Stirling number of second kind that counts the number of partitions in k classes of n objects.

In the formulas above, we have intentionally omitted parenthesis, understanding that both & and + be right associative.

Example 1. The formula A + B + C abbreviates

$$(A \wedge B \wedge C) \vee (A\&B \wedge C) \vee (A\&C \wedge B) \vee (A \wedge B\&C) \vee (A\&B\&C)$$

The following distributive axioms hold that relate classical connectives and classes.

1. (A ∨ B)&C = (A&C) ∨ (B&C)
2. (A ∧ B)&C = ((A&C) ∧ B) ∨ (A ∧ (B&C))

A clause of the form

X <-- B1 + ... + Bm

is an abbreviation for one of the following

a) if m=0 X
b) if m>0 A1&(A2&(...&(Ak&X)...)) ∨
¬(A1&(A2&(...&(Ak& (B1 + ... + Bm))...)))

for each finite multiset of atoms {[A1,A2,...,Ak]} (compound brackets {[and]} enclose multiset elements).

The intuitive meaning of the clause

X <-- B1 + ... + Bm (*)

is that all the atoms occurring in class X must synchronize to be replaced with the body B1 + ... + Bm. Item (b) above can be better understood by considering that, if the atoms in class X occur as part of a larger class Y, they can still be replaced with B1 + ... + Bm that, in turn, will synchronize themselves with the remaining atoms of Y. On the contrary, if only some atoms of X are present in the goal, they cannot be replaced by clause (*). Hence, the symbol "&" occurring in a class does in no way be interpreted as a classical "∧", since the truth value of a class does not functionally depend on the truth values of the atoms it is composed with. We will come again on this issue in example 2 below.

A (concurrent) computation of a goal G is a sequence of goals G=G1,G2,..., where each G(i+1) is derived from Gi.

A (concurrent) refutation of G is a computation ending with the empty goal.

Given a goal G of the form

<-- G1 + ... + Gm

and a clause

A1&...&An <-- B1 + ... + Bk

we can derive a new goal g1

$$\leftarrow [B\ 1]_\lambda + \ldots + [B\ k]_\lambda + [G\sigma 1]_\lambda + \ldots + [G\sigma m]_\lambda$$

if and only if

- $n \leqslant m$

- σ is a permutation of the indexes of g and λ is a unifier such that
  $[G\sigma i]_\lambda = [Ai]_\lambda \quad i=1,\ldots,n.$

Example 2. Let us have the following ground clauses

1. A <-- D
2. A&B <-- E

and the goal

$$\leftarrow A + B + C \qquad (g)$$

The goal can be nondeterministically computed in the two following ways.

<-- A + B + C	<-- A + B + C
<-- D + B + C	<-- E + C
a)	b)

Let us examine what happens when abstract syntax is used in place of the concrete one. The goal is

$$\neg(A \wedge B \wedge C) \wedge \neg(A \wedge B \& C) \wedge \neg(A \& B \wedge C) \wedge \neg(A \& C\ B) \wedge \neg(A \& B \& C) \qquad (ag)$$

The clauses 1 and 2 will originate a denumerable set of clauses, but only the following can be applied to the goal.

1a. A $\vee \neg$D	2a. A&B $\vee \neg$E
1b. A&B $\vee \neg$D&B	2b. A&B&C $\vee \neg$E&C
1c. A&C $\vee \neg$D&C	
1d. A&B&C $\vee \neg$D&B&C	

For simplicity sake, let us consider only computation (b), which leads to

$$\neg(E \wedge C) \wedge \neg(E \& C)$$

which is expressed in concrete syntax exactly as

<-- E + C.

The result of computation (b) is a conjunction of the two clauses above since clause 2a and 2b may be applied to the third and the fifth conjuncts of the original goal, respectively. The other conjuncts can obviously be disregarded, since it is sufficient to refute a single conjunct to refute a whole conjunction.

Now we can better understand why the clause

A&B <-- E

corresponds to infinitly many clauses, each adding a finite class as "context" to A&B. The goal, when written in its abstract form (as), allows to better single out two conjuncts (the last two in (as)) which are worth to be noticed. In the first A and B are synchronized, in the other A and B are synchronized also with C. Hence, also the last conjunct (in which A&B&C occurs) must be replaced, resulting in E&C. The way + has beings defined assures that the synchronization between A&B and C is inherited by E.

Finally, remark that a clause in concrete syntax in general corresponds to infinite clauses in abstract syntax, but only a finite number of them will be actually used in a computation. The effectiveness of the definition of computation is then preserved.

Coming back to our example, notice that in computation (a) all the five conjuncts corresponding to the expansion of D + B + C will be obtained from (as). In fact, clause 1a applies to the first two conjuncts of (as), and 1b-d to exactly one of the remaining conjuncts.

3. MODEL-THEORETICAL AND FIXPOINT SEMANTICS

The construction of a Herbrand model for a set of clauses involving classes needs only to slightly change the one given by vanEmden and Kowalski [5]. The difference is related to the fact that the model of a class is not the intersection of the models of the atoms that occur in it. If so, "&" would be nothing more than the classical "∧", thus vanishing our proposal to describe a synchronization mechanism.

Par abus de langage, we will call Herbrand base for a program S the set of all multisets of ground atoms

$$\{[P_1^n(t_{11},\ldots,t_{1n}),\ldots,P_k^m(t_{k1},\ldots,t_{km})]\}$$

where:
- P_i are predicate symbols occurring in S,
- J is the rank of P_i^J,
- t_{rs} are ground terms.

A Herbrand interpretation is any subset of the Herbrand base.

Given a Herbrand interpretation I:

i) a ground class X is TRUE under I if and only if the multiset of its atoms belongs to I;

ii) a conjunction of ground clauses C1∧...∧Cm is TRUE under I if and only if all Ci's are TRUE under I;

iii) a disjunction of ground (both positive and negative) classes X1∨...∨Xm is TRUE under I if and only if at least one Xi is TRUE under I;

iv) the negation of a ground class ¬X is TRUE under I if and only if X does not belong to I;

v) a universally quantified clause C is TRUE under I if and only if all its ground instances are TRUE under I.

A Herbrand model of a program S is any interpretation under which all the clauses of S are TRUE.

The semantics of a program S is the minimal Herbrand model of S, which results to be the intersection of all the Herbrand models of S.

Note that the above definition of truth values of a formula under an interpretation is given in terms of abstract syntax only. Extending it to concrete syntax is an easy task. Let us simply give here the extension in the case of clauses.
A ground clause X <-- B1+...+Bm is TRUE under I if and only if for each finite multiset of ground atoms {[A1,...,Ak]}, k≥0, the disjunction

$$A1\&(A2\&(...\&(Ak\&X)...))\vee \neg(A1\&(A2\&(...\&(Ak\&\ (B1 + ... + Bm))...)))$$

is TRUE under I.

The definition of the fixpoint semantics for a program S in abstract syntax is quite standard.

The set of interpretations of a program S is partially ordered by standard set inclusion.

Given an interpretation I for a program S, the continuous transformation T associated to S yields a new interpretation I'. I' contains the multiset of ground atoms of a class X1 if

and only if there exists a ground instance of a clause of S

$$X1 \vee \neg X2 \vee \ldots \vee \neg Xn \qquad n>0$$

and the multiset of ground atoms of each Xi, i=2,...,n, belongs to I.

As usual, an interpretation I is closed under a transformation T if and only if I contains T(I).

The semantics of a program S is the intersection of all the closed interpretations of S, which can be easily proved to be the fixpoint of the above defined continuous transformation T.

The following theorem holds.

EQUIVALENCE THEOREM.
The operational, model-theoretic and fixpoint semantics are all equivalent.

The proof of the theorem relies on the following lemmas.

LEMMA 1.
The model theoretic semantics is equivalent to the fixpoint semantics.

This lemma is a corollary of the more general theorem stating that the set of the Herbrand models of a program S is equal to the set of all the interpretations closed under the continuous transformation T associated to S.

LEMMA 2.
The operational semantics is equivalent to the fixpoint semantics.

This lemma can easily be proved, since when there is a refutation of a program S and a ground class X, the multiset of ground atoms of X belongs to the fixpoint of the transformation T associated to S.

4. CONCURRENT PROGRAMS AND PROLOG PROGRAMS

We will now briefly discuss the relationships between a pure PROLOG program and a concurrent program in which classes occur. Actually, for each concurrent program there exists an equivalent PROLOG program which is denumerably infinite.

As defined above, a clause of the form

X <-- B1 + ... + Bm

corresponds to a denumerable set of clauses

A1&(A2&(...&(Ak&X)...)) ∨
¬(A1&(A2&(...&(Ak& (B1 + ... + Bm))...)))

each Ai being an atom.
Let us now translate a clause in which classes occur into a pure PROLOG clause, i.e. let us translate classes.

First, a total ordering relation > is imposed on the predicate symbols. Then, the class

$A1(t11,...,t1n_1)\&...\&Ak(tk1,...,tkn_K)$

where A(i+1)>Ai for all i=1,...,k-1, is translated into the single atom

$Q(t11,...,t1n_1,t21,...,tkn_k)$

where Q belongs to a denumerable infinite set of new predicate symbols. The translation function must be a bijection.

Note that the rank of Q is determined as the sum of the ranks of all the Ai's occurring in the class. For instance, the class

A1(x,y)&A2(x,z,w)

is translated into the following atom

Q(x,y,x,z,w).

Notice also that the condition on the ordering among atoms in a class is not a restriction, since the relative position of atoms in a class is both syntactically and semantically irrelevant.

The following fact is obviously true.

FACT. Given a translation from classes to atoms and two classes X and Y unifiable by λ, the translations of X and Y are still unifiable by λ.

We will now show that a concurrent computation of a goal is equivalent to a finite set of PROLOG computation. As mentioned above, the infinity of the translated program does not affect the effectiveness of the computations, because only a finite number of the clauses obtained by translation will actually be used in a computation.

Recall that a concurrent goal

<-- B1 + ... + Bm

corresponds to the following conjunction of PROLOG goals (let Bi be the translation of atom Bi, Qij the translation of the ij-th class, Q the translation of B1&...&Bm).

(<-- B1 ∧ ... ∧ Bm) ∧
(<-- Q11 ∧ ... ∧ Q1k_1) ∧
...
(<-- Qp1 ∧ ... ∧ Qpk_p) ∧
<-- Q

A step in a concurrent computation of a goal is then equivalent to a step of standard PROLOG computation on suitable selected goals coming from the translation. Of course, these must contain an instance of the header of the clause to be applied. Needless to say, a concurrent refutation corresponds to a set of PROLOG computations, one of which is a refutation.

The above remarks allow us to state the following theorem.

COMPLETENESS THEOREM
Any unsatisfiable (i.e. having no model) set consisting of a concurrent goal and a concurrent program has a refutation.

5. AN EXAMPLE

In order to illustrate the expressive power of our proposal, let us write a program that implements a "semaphore", through which a set of jobs can be synchronized. The program consists of four clauses defining the two classical primitives on semaphores p and v, and of two clauses implementing a queue.

```
1. p(sem_id,Job_id)&sem(sem_id,0,q)
            <-- enqueue(Job_id,q,q') + sem(sem_id,0,q')
2. p(sem_id,Job_id)&sem(sem_id,s(n),NIL)
            <-- sem(sem_id,n,NIL) + ack(Job_id)

3. v(sem_id,Job_id)&sem(sem_id,0,Job_id'.q)
            <-- sem(sem_id,0,q) + ack(Job_id) + ack(Job_id')
4. v(sem_id,Job_id)&sem(sem_id,n,NIL)
            <-- sem(sem_id,s(n),NIL) + ack(Job_id)

5. enqueue(Job_id,NIL,Job_id.NIL) <--
6. enqueue(Job_id,Job_id'.q,Job_id'.q')
            <-- enqueue(Job_id,q,q')
```

Natural numbers are represented by 0 and successor (s); queues by lists ending with NIL (the empty queue); semaphores by their name, a natural number variable and a queue. Semaphores are handled through p and v. A job job_id calling p on a semaphore sem_id is allowed to proceed running if the value of the semaphore (the second argument of sem_id) is not 0. Otherwise it is stopped and its identifier is enqueued. A job calling v either (re)starts a stopped job, if any, and dequeues its identifier, or increments the semaphore value. In both cases the calling job is resumed by sending it an acknowledgement (the definition and use of clauses ack is not shown here).

While clauses 5 and 6 are quite standard, clauses 1-4 are concurrent. Note that processes p (or v) and sem share the variable sem_id, and synchronize by communicating through it. This example shows that this kind of interaction, and also more complicated ways of synchronous communication, can be naturally and explicitly described by having more than one atom in a clause header. In fact, the specification of process sem, that manages the value and the queue of any semaphore, is isolated from those processes (p and v) that actually exploit the semaphore mechanism.

6. CONCLUSIONS

We have defined a first order semantics for an extension to PROLOG, based on a synchronization and communication primitive. The expressive power of the resulting language is stronger than the one of PROLOG. An intuitive argument to this claim can be found in the fact that a program involving such a feature corresponds to a denumerable infinite set of pure clauses. Furthermore, standard PROLOG programs can be structured as modules, and the possibly concurrent interactions among them can be naturally described in terms of the above primitive.

A similar solution to the problem of expressing concurrent programs in logic has been presented by Monteiro [8]. In his proposal, PROLOG is extended with the concept of event, thus leading to a temporal logic programming language.

Our future work will concern the possibility of introducing a sequential operator, following [7], and of giving it a precise logic meaning. Furthermore, we intend to enrich concurrent programs with the capability of processing infinite streams of data, as done in [1]. Finally, it is worth investigating on a concept of module that provide mechanisms to encapsulate logic programs.

REFERENCES

1. Bellia,M., Dameri,E., Degano,P., Levi,G., and Martelli,M. Applicative communicating processes in First Order Logic. Proc. 5th Int. Symposium on Programming, Torino 1982, Springer Verlag LNCS, 1-14.

2. Colmerauer,A., Kanoui,H., Pasero,R., and Roussel,P. Un sistéme de communication homme-machine en français. Groupe d'Intelligence Artificielle, Universite' d'Aix-Marseille, Luminy (1972).

3. Degano,P. Una classe di schemi ricorsivi non-deterministici paralleli. Calcolo, 14 (1977), 97-119.

4. Diomedi,S. Sulle basi teoriche di comunicazione e concorrenza nei linguaggi basati sulla logica. Tesi di laurea, Dip. di Informatica, Univ. di Pisa, 1983.

5. vanEmden,M.H., and Kowalski,R.A. The semantics of predicate logic as a programming language. J.ACM 23, 1976, 733-742.

6. Kowalski,R.A. Logic for Problem Solving. Artificial Intelligence Series, N.J. Nilsson ed., North-Holland, 1979.

7. Monteiro,L.F. A Horn-like logic for specifying concurrency. Proc. 1st Int. Logic Programming Conf., Marseille (1982), 1-8.

8 Monteiro,L.F. A proposal for distributed programming in logic. Unpublished manuscript.

9. Pereira,L.M. and Monteiro,L.F. The semantics of parallelism and co-routining in logic programming. COLLOQUIA MATHEMATICA SOCIETATIS JANOS BOLYAI, no. 26, North-Holland, Amsterdam (1981), 611-657.

This work has been partially supported by Ministero della Pubblica Istruzione.

ON COMPILING PROLOG PROGRAMS ON DEMAND DRIVEN ARCHITECTURES.

Bellia M.(*), Levi G.(*), Martelli M.(+)

(*) Dipartimento di Informatica
University of Pisa (Italy)

(+) CNUCE Institute of CNR
Pisa (Italy)

ABSTRACT

A compiler is proposed that maps Prolog clauses into a language (LCA/1) with clauses annotated according to functional dependencies. LCA/1 has a demand driven computation rule and allows to cope with streams and lazy constructors.

The compilation eliminates the non-determinism related to the choice of the literal to compute and guarantees an efficient computation.

1. Introduction.

Non-determinism in Prolog comes in two flavours [1]. The first one is related to the full declarative programming style and comes from the absence of any ordering in the literals occurring both in the clause right-part and in the goal. The second one is related to the relational calculus and comes from the existence of superposable clauses (i.e. clauses whose left parts atomic formulas are unifiable).

Both of the above features contribute to making Prolog a milestone of the logic based programming languages and, at the same time, the basis for all the applications where calculus and reasoning merge: expert systems, relational knowledge base management, software systems specifications and various A.I. applications are only some of them [2,3,4].

Nevertheless, all these powerful Prolog aspects cause a high complexity in the Prolog run-time support because a non accurate choice of the literal to be computed can make highly non-deterministic even potentially deterministic computations. This is a direct consequence of the first type of non-determinism because Prolog programs do not explicitly state for each variable which literals "compute" the value and which literals use such a value.

Obviously, specific interpreters choose particular strategies such as the left to right evaluation of the literals, but this is a very strict choice and does not solve the problem. Incidentally, it is worth to note that this kind of complexity cannot be reduced by running programs on efficient and Prolog oriented machines.

In order to avoid the first type of non-determinism and to speed up the computation of those relations which are (multi-output) functions, many authors [5,6,7] have experimented control languages to attach algorithms to Prolog programs [8]. The authors have considered some logically based functional languages [9,10] and defined a functional logic language, LCA [11], which is a clause language with terms constrained to be either input or output terms. LCA could integrate Prolog, as an algorithmic component which allows to explicitly express programs involving functions and to compute them in a simple and efficient way.

Nevertheless, all the proposed solutions are partially inadequate. In fact all of them loose transparency with respect to declarativeness: i.e. the resulting programs contain procedural features.

Our aim is:

- to save the Prolog expressive power with its uniform view of relations and functions;
- to develop a technique for automatically eliminating the first type of non-determinism by attaching algorithms to clauses.
- to develop an efficient interpreter able to compute the intermediate form obtained with the above step.

The basic idea to achieve this goal is to define a language

(LCA/1, a generalization of LCA) whose programs are sets of "fully annotated" (Horn) clauses. Full annotation means that all the variables (not the terms) occurring in a clause (or goal) are annotated as INput or OUTput variables. Different occurrences of the same variable are possibly annotated in different ways. The "fully annotated" clauses must obey some syntactic constraints ensuring that each OUT variable can be computed in exactly one way.

The language interpreter has been defined along the lines of the interpreter already given for LCA. Its main features are:

- a demand driven computation rule;
- the ability to handle lazy data constructors;
- the ability to handle only the second (and really semantic) type of non-determinism.

The second step is to define a translator from Prolog programs into fully annotated programs. The translator associates to each clause of a Prolog program a set of fully annotated clauses. Each of them expresses both the specific state that the variables in a goal must satisfy in order to apply the clause (i.e. the variables which are already bound or not), and a specific functional dependency among the atomic formulas (i.e. which computes what). All the fully annotated clauses, associated to each clause, only depend upon the variables occurring in the clause and are not superposable.

The compilation of Prolog programs onto a demand driven machine seems a promising solution to save on one hand, all the features of Prolog programming and, on the other hand, to earn the efficiency of running programs on a demand driven architecture.

Section 2 will give a brief introduction to LCA/1. Sections 3 and 4 treat the translation in detail, while section 5 will describe the LCA/1 interpreter.

2. The LCA/1 language.

In this section we will not describe all the details of LCA/1, because it is quite similar to other proposals [11], but we will point out the main differences.

The first one is that in a term the occurrence of a variable symbol x is always annotated by IN or OUT. We call these terms fully annotated data terms and we refer to variables annotated by IN (OUT) as input (output) variables.

The atomic formula will contain only fully annotated data terms.

Let us introduce some definitions:

- constant term: a term without variables;
- input term: a term with input variables only;
- output term: a term with at least one output variable.

The following are examples of fully annotated clauses:

$$
\begin{array}{ll}
*(s(x_{IN}), y_{IN}, z_{OUT}) & \leftarrow\!\!- *(x_{IN}, y_{IN}, w_{OUT}), +(w_{IN}, y_{IN}, z_{OUT}) \\
*(s(x_{IN}), y_{IN}, z_{IN}) & \leftarrow\!\!- *(x_{IN}, y_{IN}, w_{OUT}), +(w_{IN}, y_{IN}, z_{IN}) \\
*(S(x_{IN}), y_{OUT}, z_{OUT}) & \leftarrow\!\!- *(x_{IN}, y_{OUT}, w_{OUT}), +(w_{IN}, y_{IN}, z_{OUT}) \\
REV(x_{IN} \cdot y_{IN}, w_{OUT}) & \leftarrow\!\!- REV(y_{IN}, z_{OUT}), APP(z_{IN}, x_{IN} \cdot nil, w_{OUT}) \\
REV(x_{IN} \cdot y_{OUT}, w_{IN}) & \leftarrow\!\!- REV(y_{OUT}, z_{IN}), APP(z_{OUT}, x_{IN} \cdot nil, w_{IN})
\end{array}
\qquad (*)
$$

where s and . are function symbols and *, +, REV and APP are predicate symbols.

The predicate * holds if the third argument is equal to the product of the first two arguments, and the predicate REV holds if the first argument is the reverse list of the second argument. Moreover, the intended meaning of the first clause of * is that, for any x and y, the result of the product of s(x) and y is the sum of y with the product of x and y, while the second clause of * means that for any triple of numbers x, y and z, z is the result of the product of s(x) and y if z is the sum of y with the product of x and y.

Examples of fully annotated goals are the following:

$\leftarrow\!\!-\ *(s(s(0)), S(0), x_{OUT})$

$\leftarrow\!\!-\ *(s(s(x_{IN})), s(y_{OUT}), z_{OUT}), +(s(s(0)), x_{OUT}, s(s(s(0))))$

$\leftarrow\!\!-\ REV(a.b.c.nil, z_{OUT})$

$\leftarrow\!\!-\ REV(a.x_{OUT}.c.nil, c.b.a.nil)$

The syntax of the language has to satisfy some constraints to have the desired properties. In the following, we assume familiarity with the terminology and the notation used in [1].

Let $M_{IN}(a)$ ($M_{OUT}(a)$) be the multiset of the input (output) variables of an atomic formula a.

Let H<-- a1,a2,...,an be a clause, where H is the conclusion atomic formula, the ai's are the atomic conditions, and all the atomic formulas are fully annotated (a1,a2,...,an can also indicate a goal).

Condition 1.

1.1) For each clause and for each ai, $M_{IN}(H) \cap M_{OUT}(ai) = \emptyset$.

1.2) For each clause or goal the multiset $\bigcup_{i \in [1,n]} M_{OUT}(ai)$ must be a set.

This condition ensures that every variable is computed in exactly one way by only one atomic formula.

Condition 2.

For each atomic formula ai in a clause or goal, each variable belonging to $M_{IN}(ai)$ must belong to $M_{OUT}(ak)$ (or to $M_{IN}(H)$ in the case of a clause), where ak is an atomic formula of the clause or goal such that $i \neq k$.

This condition forbids to have atomic formulas whose input variables do not occur as output variables of other atomic formulas.

Condition 3.

The multiset $M_{IN}(H)$ must be a set.

This condition is complementary to Condition 1 (about the uniqueness of the computations), and forbids to put conditions on the input variables of the conclusion atomic formula; i.e. the unification process does not need to control equality on the input variables. This allows to have a simple and (possibly) parallel unification algorithm.

Constraints on the values computed by different variables are allowed and efficiently handled by the primitive predicate EQp. The semantics of EQp corresponds to the Prolog assertion:

$$EQp(\underbrace{x,\ldots,x}_{p+1\text{-times}}) \leftarrow\!- \square \qquad (EQ1(x,x) \leftarrow\!- \square).$$

Note that because of Conditions 1,2 and 3 all variable symbols occurring in $M_{OUT}(H)$, must belong either to $M_{IN}(H)$ or to $M_{OUT}(ai)$, for some ai in the clause, or must not occur in the clause right part.

As a consequence, any output variable is either computed by one atomic formula only or must be considered bound to the set of all the terms of the Herbrand Universe.

LCA/1 is a generalization of LCA [11] mainly motivated by the compilation of Prolog clauses. Such a generalization is obtained by redefining the term structure and by relaxing some constraints of LCA. Nevertheless, the main properties of the LCA semantics are saved in the operational semantics of LCA/1. Thus, the definition of the LCA/1 interpreter is structurally similar to the one defined in [11]. Section 5 briefly analyses the external evaluation rule and the new formulation of the computation rule needed to handle full annotations.

3. The compiler.

The compiler from Prolog into LCA/1 is a mapping of clause structures of Prolog into LCA/1 ones.

This mapping is based on the concept of state of the computation, i.e. the state of the variables during the computation of the current goal: each variable can be already bound (totally or partially computed) or not. The variable can be considered, in the first case, as a possible input and, in the second case, as a possible output for an atomic formula.

A second aspect of the concept of state is related to the applicability of a clause. LCA/1 allows to explicitly define, for each conclusion atomic formula, which variables are assumed to be input (and thus must be bound to a value by the unification), and which variables are assumed to be output (and will have a value "computed" by the clause) at resolution time.

The first aspect of state is also present in Prolog (bound and unbound variables in the unification process).

The main idea of the transformation is that a Prolog atomic formula implicitly expresses a finite number of possible

different states (the second aspect) and this number combinatorially depends upon the number of variables occurring in the clause. A Prolog clause can then be mapped into a set of fully annotated clauses, each of them expressing a particular state.

As an example of the transformation, the three clauses in (*) are some of the eight fully annotated clauses defined by the following Prolog clause:

$*(s(x),y,z) \leftarrow\!- *(x,y,w),+(w,y,z)$.

Let us take the first clause of (*), i.e.:

$*(s(x_{IN}),y_{IN},z_{OUT}) \leftarrow\!- *(x_{IN},y_{IN},w_{OUT}),+(w_{IN},y_{IN},z_{OUT})$.

This clause explicitly defines a state of applicability, where the variables x and y must be bound and where the variable z is computed by the the clause itself.

4. The transformation.

In order to formally define the transformation from PROLOG programs into LCA/1 programs we will use the following simple structures.

DEFINITION 1 (variable sequence or sequence).
To each term t we can associate the variable sequence containing all the variable occurrences as found by a pre-order term traversing process.

As an example, $\langle x,y,x,z\rangle$ is the sequence associated to the term $f(x,g(y,x),z)$.
If t is a constant term, the sequence associated to t is the empty sequence. Let s be the sequence of length n associated to the term t, s[i] (or t[i]), for each $i \in [1,n]$, selects the i-th variable in s.
In the following, the concept of sequence will be generalized to atomic formulas by associating to each formula of the form P(t1,...,tk) the sequence obtained by concatenating the sequences s1,...,sk associated to the terms t1,...,tk respectively.

DEFINITION 2 (annotated sequence).
Let s be a sequence of length n, we define $\{IN,OUT\}^s$ as the set of all the annotated sequences generated by s.

The annotated sequence $v \in \{IN,OUT\}^s$ differs from s because, for each $i \in [1,n]$, v[i] is the variable s[i] annotated by IN or by OUT. We call v[i] an annotation for the variable s[i].
The set $\{IN,OUT\}^s$ contains exactly 2^n annotated sequences.

DEFINITION 3 (substitution sequence).
Let s be a sequence and v be an annotated sequence of the same length of s, we define a substitution as the pair (s,v).

DEFINITION 4 (substitution applicability).
Let t be a term and S be the substitution (r,v), we say that S is applicable to t iff r is equal to the sequence associated to t.
The application of S to the term t results in the term t' such that:

$$\forall i \in [1,n], \quad t'[i] = v[i],$$

if n is the length of s.

The transformation maps a clause c into a set U(c) of annotated clauses. It will be described in a two step process. First of all, given a clause c of the form H<--L, we compute the set U'(c) of partially annotated clauses. The clauses in U'(c) have all the variables occurring in H replaced by annotated variables. In the first step, the local variables of c (i.e. variables not occurring in the clause conclusion) are ignored.
The second step takes care of the local variables by providing the computation of a fully annotated clause for each clause in the set U'(c). In the same way, the second step is able to provide the transformation of a goal statement into the corresponding fully annotated goal.

4.1 The computation of U'(c).

Let c be the clause H<--l1,...,ln, the computation of U'(c) proceeds as follows:

1) Define $\{IN,OUT\}^s$, where s is the variable sequence associated to H.

2) Compute the subset $K \subseteq \{IN,OUT\}^s$ which contains all the annotated sequences having multiple occurrences of the same variable annotated by IN. Note that, the set K could be empty. The set is empty if and only if the sequence s does not contain multiple occurrences of the same variable.

3) $\forall r \in \{IN,OUT\}^s - K$, let (s,r) be a substitution. Compute H'<--L' U'(c) as follows:
   + H' is the atomic formula resulting from the application of (s,r) to H;
   + L' is the sequence l1',...,lm' such that:
     - n <= m, and
     - $\forall i \in [1,n]$, and for each variable x occurring both in li and in the sequence s, li' contains x annotated as follows:
       1) if x occurs in r annotated by IN, then each occurrence of x is replaced in li' by x_{IN}.
       2) if x occurs in r only annotated by OUT, then one of the following holds:
          a) $\exists j \in [1,n]$ such that $i \neq j$ and lj' already contains an occurrence of x_{OUT}. Then each occurrence of x is replaced in li' by x_{IN}.

b) $\forall j \in [1,n]$, such that $i \neq j$, lj' does not contain occurrences of x_{OUT}. Then
   1) if li contains exactly one occurrence of x, then the occurrence of x is replaced in li' by x_{OUT}.
   2) if li contains $p+1$ occurrences of x, then
      + the first occurrence of x is replaced in li' by x_{OUT} and all the other occurrences are replaced by different renamings of x annotated by OUT.
      + let $x1_{OUT},\ldots,xp_{OUT}$ be the above introduced renamings. Then
        $$EQp\ (x_{IN}, x1_{IN},\ldots,xp_{IN})$$
        is a <u>special</u> atomic formula lu' in L' for some $u \in [n+1,m]$.

4) $\forall r \in K$, we add to the set resulting from step 3) the clause

$$H'\text{<--}l1',\ldots,ln',\ldots,lh',\ldots,lm' \qquad (n <= h < m)$$

obtained as follows:
+ for each variable x occurring in r more than once, let $x1_{IN},\ldots,xp_{IN}$ be a renaming for each occurrence but the first. Then
  $$EQp\ (x_{IN}, x1_{IN},\ldots,xp_{IN})$$
  is an atomic formula lu' for some $u \in [h+1,m]$
+ let r' be the annotated sequence r whose variables are renamed according to the above step, then (s,r') is still a substitution and $H'\text{<--}l1',\ldots,ln',\ldots,lh'$ is the result of step 3) applied to (s,r').

4.2 Example.

Let us consider the clause c:

$$A(x,d(x)) \text{ <-- } B(x,y),E(x,x)$$

where d is a function symbol, A,B and E are predicate symbols and x, y are the variables occurring in the clause such that y only is local. Then, the computation of $U'(c)$ proceeds as follows:

1) $s=\langle x,x\rangle$
   $\{IN,OUT\}^s = \{\langle x_{IN},x_{IN}\rangle,\langle x_{IN},x_{OUT}\rangle,\langle x_{OUT},x_{IN}\rangle,\langle x_{OUT},x_{OUT}\rangle\}$

2) $K=\{\langle x_{IN},x_{IN}\rangle\}$

3) $\forall\ s \in \{\langle x_{IN},x_{OUT}\rangle,\langle x_{OUT},x_{IN}\rangle,\langle x_{OUT},x_{OUT}\rangle\}$
   + $s=\langle x_{IN},x_{OUT}\rangle$
     $H'=A(x_{IN},d(x_{OUT}))$
     $L'=B(x_{IN},y),E(x_{IN},x_{IN})$
   + $s=\langle x_{OUT},x_{IN}\rangle$
     $H'=A(x_{OUT},d(x_{IN}))$
     $L'=B(x_{IN},y),E(x_{IN},x_{IN})$
   + $s=\langle x_{OUT},x_{OUT}\rangle$

$H' = A(x_{OUT}, d(x_{OUT}))$
$L' = B(x_{OUT}, y), E(x_{IN}, x_{IN})$
or
$L' = B(x_{IN}, y), E(x_{OUT}, x1_{OUT}), EQ1(x_{IN}, x1_{IN})$

4) $\forall r \in \{<x_{IN}, x_{IN}>\}$
+ $EQ1(x_{IN}, x1_{IN})$
+ $r' = <x_{IN}, x1_{IN}>$
$H' = A(x_{IN}, d(x1_{IN}))$
$L' = B(x_{IN}, y), E(x_{IN}, x_{IN}), EQ1(x_{IN}, x1_{IN})$

The computation defines two U'(c), each one containing four fully annotated clauses, which differ in the right part of the clause obtained from the substitution $s = <x_{OUT}, x_{OUT}>$.

4.3 Remarks about U'(c)

Proposition 1
For each Prolog clause c, U'(c) contains at least one clause. Moreover, U'(c) contains exactly the clause c iff the conclusion atomic formula of c has no variables.

Proposition 2
U'(c) as computed by steps 1)-4) is not unique. In fact, step 3.2) could lead to more than one U'(c), if more than one atomic formula in the right part contains a variable which, in the sequence r, is only annotated by OUT.
Actually, we are not concerned with the choice of U'(c), although the problem of choosing the best atomic formula is the key issue for optimization.

Proposition 3
The following properties hold for the annotations of the clauses in U'(c):

Property 1 No conclusion atomic formula contains more than one occurrence of the same variable annotated by IN, as guaranteed by the subset K in steps 2) and 4).

Property 2 No clause right part contains more than one occurrence of the same variable annotated by OUT, as guaranteed by step 3).

Property 3 For each clause whose conclusion atomic formula contains a variable annotated by OUT, only one of the following cases holds:
1-the same variable annotated by IN occurs in the conclusion atomic formula also;
2-the same variable annotated by OUT occurs in exactly one atomic formula in the clause right part;
3-the same variable annotated by OUT occurs in the conclusion atomic formula only.
This property is guaranteed by the variable renamings

introduced in point 2.b.2 of step 3).

Property 4 For each variable annotated by IN in a clause atomic formula, only one of the following cases holds:
- the same variable annotated by IN occurs in the clause conclusion;
- the same variable annotated by OUT occurs in exactly another atomic formula of the clause right part.

This property is guaranteed by point 1) and 2.a) of step 3).

4.4. The computation of U(c)

The computation of U(c) provides the annotation of the local variables occurring in c. Local variables are variables which occur only in the right part of the clause. Such variables are left unchanged by the computation of U'(c). Thus the following property holds:

Proposition 4
$\forall \bar{c} = H \leftarrow\!- L$, $\bar{c} \in U'(c)$ iff there exists L' such that:
$H \leftarrow\!- L' \in U(c)$.

Thus, in order to obtain U(c), for each clause $\bar{c}$ of U'(c), only L' has to be computed.

Let $H \leftarrow\!- l1,\ldots,ln$ be a clause in U'(c), then U(c) contains $H \leftarrow\!- l1',\ldots,lm'$ $(n<=m)$ such that:

5) $\forall i \in [1,n]$ such that li is already a fully annotated atomic formula (i.e., li does not contain local variables) then $li'=li$;

6) Let $i \in [1,n]$ be such that li contains at least a local variable x, then one of the following cases holds:
   1) $\exists j \in [1,n]$, such that $i \neq j$ and lj' contains an occurrence of x_{OUT}. Then each occurrence of x is replaced in li' by x_{IN}.
   2) $\forall j \in [1,n]$, such that $i \neq j$, lj' does not contain occurrences of x_{OUT}, then:
      a) if li contains exactly one occurrence of x, then the occurrence of x is replaced in li' by x_{OUT}.
      b) if li contains p+1 occurrences of x, then the following steps are performed:
         - the first occurrence of x is replaced in li' by x_{OUT} and all the other occurrences by renamings of x annotated by OUT.
         - let $x1_{OUT},\ldots,xp_{OUT}$ be the above introduced annotated renamings for x. Then
           $EQp(x_{IN}, x1_{IN},\ldots,xp_{IN})$
           is the atomic formula lu', for some $u \in [n+1,m]$, added to the right part of the transformed clause.

4.5 Example

As an example of computation, let us consider the computation of U(c1) and U(c2) in the case of the following predicates for the addition:

c1: $+(0,y,y)$ <--
c2: $+(s(x),y,s(z))$ <-- $+(x,y,z)$

where s is the successor function.

$$U(c1)=\{+(0,y_{IN},y1_{IN}) \leftarrow EQ1(y_{IN},y1_{IN}) \quad (1)$$
$$+(0,y_{IN},y_{OUT}) \leftarrow \quad (2)$$
$$+(0,y_{OUT},y_{IN}) \leftarrow \quad (3)$$
$$+(0,y_{OUT},y_{OUT}) \leftarrow \} \quad (4)$$

$$U(c2)=\{+(s(x_{IN}),y_{IN},s(z_{IN})) \leftarrow +(x_{IN},y_{IN},z_{IN}) \quad (5)$$
$$+(s(x_{IN}),y_{IN},s(z_{OUT})) \leftarrow +(x_{IN},y_{IN},z_{OUT}) \quad (6)$$
$$+(s(x_{IN}),y_{OUT},s(z_{IN})) \leftarrow +(x_{IN},y_{OUT},z_{IN}) \quad (7)$$
$$+(s(x_{IN}),y_{OUT},s(z_{OUT})) \leftarrow +(x_{IN},y_{OUT},z_{OUT}) \quad (8)$$
$$+(s(x_{OUT}),y_{IN},s(z_{IN})) \leftarrow +(x_{OUT},y_{IN},z_{IN}) \quad (9)$$
$$+(s(x_{OUT}),y_{IN},s(z_{OUT})) \leftarrow +(x_{OUT},y_{IN},z_{OUT}) \quad (10)$$
$$+(s(x_{OUT}),y_{OUT},s(z_{IN})) \leftarrow +(x_{OUT},y_{OUT},z_{IN}) \quad (11)$$
$$+(s(x_{OUT}),y_{OUT},s(z_{OUT})) \leftarrow +(x_{OUT},y_{OUT},z_{OUT})\} \quad (12)$$

Note that U(c1) and U(c2) are unique.

4.6. Remarks about U(c).

Because of Proposition 4, some properties, already given for the set U'(c), hold for the set U(c) as well. In the following, we state the properties which hold for the set U(c) and we show how the clauses in U(c) satisfy the conditions given for the annotated clauses of LCA/1.

Proposition 1'
Proposition 1 holds in the case of U(c) also. However, U(c) could contain exactly one clause c', such that $c \neq c'$, depending on the occurrence of local variables in c.

Proposition 2'
Proposition 2 holds in the case of U(c) also. In fact, in addition to the non-uniqueness of U'(c) (caused by step 3.2), step 6) could hold for more than one U(c) for similar motivations. The remarks given about U'(c), concerned with the choice of the best atomic formula, apply to U(c) as well.

Proposition 3'
Properties of U'(c), involving only the clause conclusion, obviously hold even for U(c), namely properties 1 and 3 of Proposition 3. In addition, clauses in U(c) satisfy Properties 2 and 4 because of steps 5) and 6).

We will now show that, if Proposition 3' holds, the conditions given for the clauses of the language introduced in Section 2 are satisfied.

- Condition 1 Point 1 is achieved by property 4' of Proposition 3', because, when it is applied to the conclusion atomic formula, the first case holds. Point 2 is guaranteed by property 2' of Proposition 3'.

- Condition 2 The condition immediately follows from property 4' of Proposition 3'.

- Condition 3 The condition is guaranteed by property 1' of Proposition 3'.

As a final remark let us note that property 3 of Proposition 3 is not essential and follows directly from the other properties in the Proposition.

Finally, a few words about the goal. A goal is a special clause structure whose left part is "empty", and, thus, it only has local variables. The computation of U'(c), in the case of a goal c, is the set {c}. Given U'(c)={c}, the computation of U(c) proceeds as in the case of any other clause structure. It results in the set {c'} whose unique clause is a fully annotated goal and satisfies all the above propositions.

4.7 Example

As an example of a goal computation let us consider the following clause c:

<-- +(3,u,v)

The computation of c is:

$$U(c) = \{\text{<-- } +(3, u_{OUT}, v_{OUT})\}$$

Note that the solution is unique.

5. The language interpreter.

While mentioning the language features, we pointed out in Section 2 how LCA/1 is a generalization of LCA, proposed for functional (even if non-deterministic) computations in Prolog-like programming enviroments. Thus the language interpreter we propose is defined along the same lines of the LCA interpreter given in [11]. It has a similar algebraic definition and it handles some features, like lazy constuctors and streams, in exactly the same way.

Nevertheless, some relevant differences must be considered, mainly with respect to:

- the evaluation order of the goal atomic formulas;
- the clause unification mechanism.

5.1 The evaluation order and the demand driven rule.

The evaluation order of atomic formulas in a fully annotated goal is established on the basis of a demand driven rule.

Each fully annotated goal contains some of the following three types of atomic formulas:

1) constant formulas: atomic formulas whose terms are only constant terms;

2) input formulas: atomic formulas whose terms are either constant or input terms and contain at least one input term;

3) output formulas: atomic formulas containing at least one output term.

The first two types of formulas correspond to formulas which only put constraints on the goal or on the values of the variables occurring in the goal. As a matter of fact, atomic formulas, whose predicate symbol is EQp, are of the second type and their evaluation constraints the evaluation of the formulas which use the same variables. Annotations allow us to define global each variable annotated by OUT which occurs in a goal and such that:

+ the variable does not occur annotated by IN in the goal
  or
+ the variable occurs annotated by IN in input formulas only.

Thus a goal could be partitioned into two parts. One part consists of the set of all the atomic formulas which contain at least one occurrence of a global. This part provides the computations of the "results" of the goal evaluation.

The atomic formulas of the second part do not contain globals and only provide the computations of intermediate (and, possibly unessential) values.

The evaluation of a goal proceeds as follows: the constant formulas are evaluated first, then the input formulas containing globals are considered. Finally, when the goal does not contain any constant nor input formulas with globals, the output formulas which contain at least one global are evaluated.

The evaluation of an atomic formula of the second or third type could require the evaluation of output formulas including atomic formulas not containing globals. In the case of the evaluation of formulas not containing globals, input formulas are evaluated first.

Note that the order is statically defined by the input-output relation among atomic formulas. The relation is induced by the occurrence of the same variable annotated by IN and OUT respectively in different atomic formulas. The relation we have defined is a partial order. Hence the choice of the formula, where more than one choice is possible, is unessential to a right sequentialization of the computation.

5.2 The clause application mechanism

The clause application mechanism allows to apply a clause to an atomic formula in the goal, and results in the evaluation of goal atomic formulas. Whenever the value of a variable is needed to apply a clause, the atomic formula computing that variable is selected (by the Demand Driven Rule) for the evaluation.

The mechanism is mainly based on a term unification mechanism which provides:

- the binding of the input variables which occur in the conclusion atomic formula of the clause with the corresponding input or constant terms of the goal atomic formula;

- the binding of the output variables which occur in the goal atomic formula with the corresponding output or constant terms of the conclusion atomic formula of the clause.

Thus, the application of the unification to terms is not symmetric. In fact, unification behaves, on one hand, like a match of input terms in the goal atomic formula to input terms in the clause conclusion, and, on the other hand, like a match of output terms in the clause conclusion to the output terms in the goal atomic formula.

The unification of a term in the goal atomic formula, tg, with the corresponding term in the clause conclusion, tc, has the following properties.

Proposition 5

The term tg is unifiable with tc if one of the following cases holds:

1) tg is a constant term;
2) tg is an input term and tc is either an input or a constant term;
3) tg is an output term and tc is either an output or a constant term.

Proposition 6

The unification of tg and tc results in the pair of unifiers $(\lambda_{IN}, \lambda_{OUT})$ respectively for input and output variables, if and only if:

1) tg is a constant term and λ_{IN} is such that:

$$tg = [tc]_{\lambda_{IN}}$$

Note that, if tc is an output term, the unification requires the evaluation of the right part of the clause in order to compute the output variables occurring in tc.

2) tg is an input term and λ_{IN} is such that:

$$tg = [tc]_{\lambda_{IN}}$$

In this case, the unification could require the evaluation of the goal in order to compute the variables occurring as inputs in tg and corresponding to terms (different from variables) in tc.

3) tg is an output term such that:

3.1) tg is an output variable. Then

$$[tg]_{\lambda_{OUT}} = tc$$

that is, λ_{OUT} contains a binding of the variable tg to the term tc. Moreover, tc must be a constant term or an output term containing only output variables.

3.2) tg is a term of the form f(tg1, ..., tgk) (where f is a data constructor and at least one of the tgi's is an output term), then:

$$[tg]_{\lambda_{OUT}} = [tc]_{\lambda_{IN}}$$

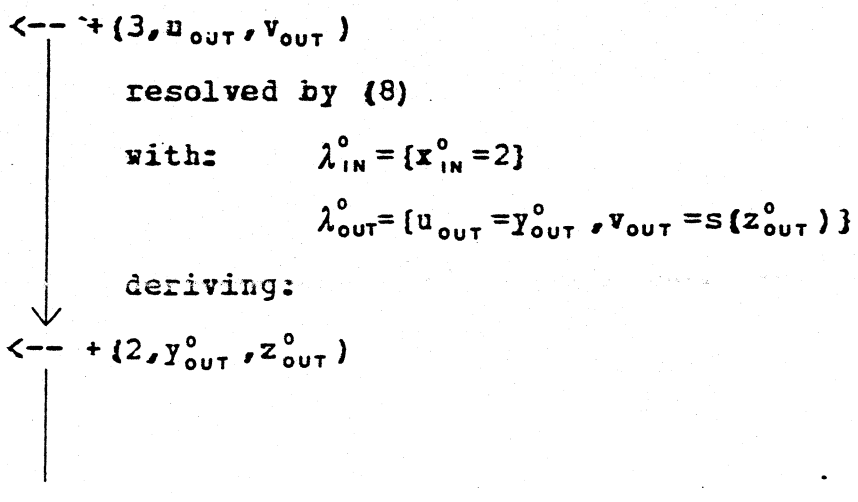
If this is the case and if tc is an output variable, the unification needs the evaluation of the right part of the clause to obtain for tc the term f(tc1, ..., tck). Then, the unification proceeds through the unification of tgi, tci for each i from 1 to k.

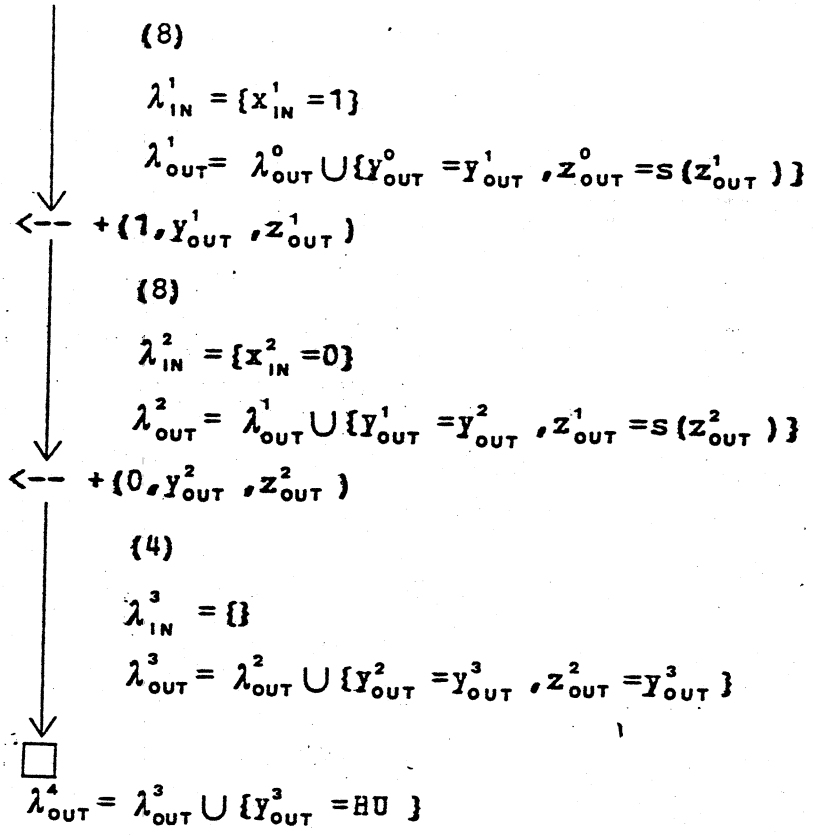
Note that, because of 3.1, if tc is an output variable, its value f(tc1, ..., tck) contains output variables only. Thus, to obtain an unification, tg must also be a term containing output variables only.

A special case arises when tc is an output variable which does not occur in the right part of the clause, i.e. there are no atomic formulas in the goal which can compute values for the variable. In this case the variable is considered bound to all terms of the Herbrand Universe, and the value of the variable is denoted by HU. The match of such a variable to an output term tg must bind the variables in tg to HU also.

5.3 Example

As an example of a computation of a fully annotated program, let us consider the evaluation of the goal in the example in 4.7 with the clauses U(c1) and U(c2) in 4.5.





6. Conclusion

The design of new machines for logic based languages, including the functional ones, requires the project of unconventional architectures oriented to efficiently handle the language computation rules.

Thus it is important to define a (small) nucleus of primitive rules which, on one hand, guarantees to express each language computation step and, on the other hand, becomes a model to tailor the language architecture.

In this trend, we have considered the selection of atomic formulas in the goals of a Prolog computation. As a matter of fact, the selection has a remarkable relevance in the Prolog implementation because:

- the selection affects the efficiency of the computations: i.e. it can cause too long computations;
- the selection requires a specific mechanism which can even affect the efficiency of the mechanism to handle the non-determinism.

Actually, the selection is handled in two different ways. The first, common to all the Prolog implementations, makes a static selection. This is achieved either by ordering the atomic formulas from left to right [12], or by using annotations [5]. The former does not cope with efficiency, while the latter looses the declarative transparency and does

not guarantees efficiency.

The dynamic handling of the selection is the second approach [13]. It allows efficient computations but requires mechanisms which are complex and hard to build.

A promising solution to this problem seems to be a compilation of the Prolog clauses into fully annotated clauses.

An annotation assigns a role to atomic formulas by distinguishing between the one which, for a given variable, must compute a value and the ones which will use that value. In this way, a functional dependency is statically imposed on the atomic formulas. Then the selection is handled by means of a demand-driven mechanism.

In addition to it, the proposed compilation allows us to reduce both the overhead of the unification mechanism (which becomes a matching mechanism) and of the computation environment (only the output terms unifiers, λ_{OUT}, must be kept).

However, some open questions can be considered.

The first is the choice of the object program when more than one is possible. The choice is semantically unessential (as we will point out in the following) and does not affect the design or the efficiency of the demand-driven mechanism. However, it is essential in order to shorten the computations.

Given a set S of Horn clauses, the choice solutions are strictly related to a selection function which guarantees, for each goal for S, a derivation (if any) with the smallest number of input clauses [14].

The use of partially annotated clauses (clauses like those occurring in U'(C)) together with the results concerning the superposition [15] seems a promising approach towards the definition of such a function.

For what the semantics is concerned, it is simple to prove that any object obtained by the compilation is semantically equivalent to the original Prolog set of clauses (source program).

The proof could ignore the problems arising from superposable clauses and show that the derivation of the fully annotated clauses is a special case of the LUSH resolution applied to Horn clauses.

Finally, the programming environment, the proposal allows to define, deserves some remarks.

Programming applications often need to integrate declarative programming with procedural one. Such an integration will allow to easily combine declarative and procedural knowledge (i.e. algorithms) and is currently been pursued by several projects, notably Robinson's LOGLISP [16].

To obtain it, attention has to be put on the integration level which must allow, on one hand, to easily merge declarative with procedural computations, and on the other hand, to maintain, as small as possible, the nucleus for the different types of computation.

LCA/1 seems a good candidate for the integration level, in particular it allows the same nucleus to compute both declarative and procedural programs. Moreover, the proposed compiler could be lightly modified in order to be applied to programs of partially annotated clauses, thus including pure Prolog programs, LCA programs and programs whose clauses contain both Prolog and LCA atomic formulas.

REFERENCES

[1] Kowalski, R.A. Predicate Logic as a Programming Language. Information Processing 74, North Holland, 1974, pp. 556-574.

[2] Kowalski, R.A. Logic for Problem Solving. Artificial Intelligence Series, N.J. Nilsson Ed., North Holland, 1979.

[3] Logic Programming. Clark K.L. and S.A. Tarnlund Eds., Academic Press, 1982.

[4] Proceedings of the 1st Int'l Logic Programming Conference. Marseille, 1982.

[5] Clark, K., McCabe, F. and Gregory, S. IC-PROLOG Language Features. In [3], pp. 253-266.

[6] Gallaire, H. and Lasserre C. Metalevel Control for Logic Programs. In [3], pp. 173-185.

[7] Pereira, L.M. and Porto, A. Intelligent Backtracking and Sidetracking in Horn Clause programs - The Theory. Departamento de Informatica, Universitade Nova de Lisboa, Rep. 2/79 CIUNL, October 1979.

[8] Kowalski, R.A. Algorithm=Logic+Control. Comm. of A.C.M., 22, 1979, pp. 424-431.

[9] Bellia M., Degano P. and Levi G. The call by name semantics of a clause language with functions. In [3], pp. 281-298.

[10] Bellia, M., Dameri, E., Degano, P., Levi, G. and Martelli, M. Applicative Communicating Processes in First Order Logic. Lecture Notes in Computer Science, 137, Springer Verlag, 1982, pp. 1-14.

[11] Bellia, M., Dameri, E., Degano, P. Levi, G. and Martelli, M. A Formal Model for Demand-driven Implementations of Rewriting Systems and its Application to Prolog Processes. I.E.I. Internal Report, IEI-B81-3, 1981.

[12] Moss, C. The comparison of several Prolog systems. Proc. of First Logic Programming Workshop, Debrecen, 1980, pp. 198-200.

[13] Pereira, L.M. and Porto, A. Selective Backtracking. In [3], pp. 107-116.

[14] Hill, R. LUSH-Resolution and its compliteness. DCL Memo No. 78, Univ. of Edimburgh, 1974.

[15] Sato, T. and Tamaky, H. Enumeration of success patterns in Logic Programs. To be presented at 10th ICALP.

[16] Robinson, J.A. and Sibert, E.E. LOGLISP: an alternative to PROLOG. Machine Intelligence No. 10, 1982, pp. 399-420.

CONTROL OF ACTIVITIES IN THE OR-PARALLEL TOKEN MACHINE

Andrzej Ciepielewski and Seif Haridi
Department of Telecommunication and Computer Systems
Royal Institute of Technology
Stockholm, Sweden

ABSTRACT

A machine model consisting of a limited number of processors, a token pool and a storage has been defined, [HaCi]. A token represents the state of a process, which in turn executes a branch in the search tree of a logic program. Tokens in the token pool correspond to processes which are ready for execution but not allocated a processor. Processors execute processes as presecribed by the tokens and create new tokens. During an Or-parallel execution the number of processes usually exceedes the number of available processors. The problem of controlling the number of activities can be divided into two subproblems: (1) controlling the traversal of the search tree and (2) prunning some branches of the search tree. The solution to (1) can been seen as a scheduling problem and will be discussed in a forthcoming paper. To solve (2) we device a mechanism for prunning the search tree, removing from the system tokens representing no longer needed computations, when only one solution to a problem or a subproblem is required. We show how the mechanism is incorporated into the Or-parallel token machine without imposing any process hierarchy or message passing. We define a translation of the extended source language programs, [Ha,HHT], into sequences of abstract machine instructions and define the interpretation cycle of a processor for the extended instruction set. Finally we discuss how the mechanism can be generalised to prun ing of the trees when at most n solutions is reguired, and for guarded clauses [ClGr,Sh].

References

[ClGr] K L Clark, S Gregory, A Relational Language for Parallel Programming, in proceedings of ACM Symposium on Functional Programming Languages and Computer Architecture, October 1981

[Ha] S Haridi, Logic Programming Based on a Natural Deduction System, PhD Thesis, TRITA-CS-8104, Royal Institute of Technology, Stockholm 81

[HaCI] S Haridi, A Ciepielewski, An Or-parallel Token Machine, in this proceedings.

[HHT] A Hansson, S Haridi, S-Å Tärnlund, Properties of a Logic Programming Language, in Logic Programming edited by K L Clark and S-Å Tärnlund, Academic Press 82

[Sh] E Y Shapiro, A Subset of Concurrent Prolog and its Interpreter, SRI International, January 1983, also to appear as TR-003 , ICOT, TOKYO.

AN OR-PARALLEL TOKEN MACHINE

Seif Haridi and Andrzej Ciepielewski
Department of Telecommunication and Computer Systems
Royal Institute of Technology
Stockholm, Sweden

ABSTRACT

achine model consisting of a limited number of processors, a token pool and storage is defined. A token represents the state of a process, which in turn ecutes a branch in the search tree of a logic program. Tokens in the token l correspond to processes which are ready for execution but not allocated a cessor. Processors execute processes as presecribed by the tokens and ate new tokens. A processor executes a compiled form of the programs. We ine the translation of programs into sequences of abstract machine instruc-ns and define the interpretation cycle of a processor.

Introduction

n clause programs can be executed in different modes without changing their ning up to termination. The most common mode is Prolog's left-to-right ection of subgoals and depth-first traversal of the search tree using back-cking. Instead of a sequential exploration of alterantive solutions, the rch tree can be traversed in parallel. This mode has been lately called parallelism. It can be implemented on a single processor [RoSi], but comes st to its right when a large number of processors is used.

goal of our research is a multiprocessor architecture for efficient execu-n of Or-parallelism. We share this goal with a growing number of research-: [CoKi], [EKM], [Po], [UmTa] and [FNM]. We have already defined an inter-ter for Or-parallelism and investigated the feasibility of using structure ring in a distributed implementation [CiHa83A,CiHa83B]. In this inter-ter, we have studied, in detail, the problem of managing simultaneously eral binding environments. The interpreter evaluates programs in their tract source form and creates a computation process for each alternative deterministic branch.
this paper we define an abstract machine model consisting of a limited ber of processors, a pool of tokens and a storage. The unlimited number of cesses in our interpreter is now mapped onto the finite number of proces-s. A processor executes a compiled form of programs. Subgoals are ected in a specific order as defined by the sequence of instructions. We cribe the translation of logic programs into sequences of the abstract hine instructions and define the semantics of the instructions. The truction set we define here is similar to that of the sequential machine cribed in [HaSa]. Finally, we discuss methods for controlling the amount parallelism and compare our machine with other proposals.

Abstract machine model

ogic program consists of an initial call and a set of relations. A rela-n consists of a number of clauses, where each clause is either an assertion an implication. An implication has a head and a body. The body is a literal a conjunction of literals.

Ex 1: The following is a program for list-permutation; it consists of two relations: p(ermute) and d(elete):

```
1.  p([],[]).
2.  p(xs,[y|ys]) ←— d(xs,y,zs) & p(zs,ys).

1.  d([x|xs],x,xs).
2.  d([x|xs],y,[x|ys]) ←— d(xs,y,ys).
```

and a possible initial call:

```
p([1,2],ys)
```

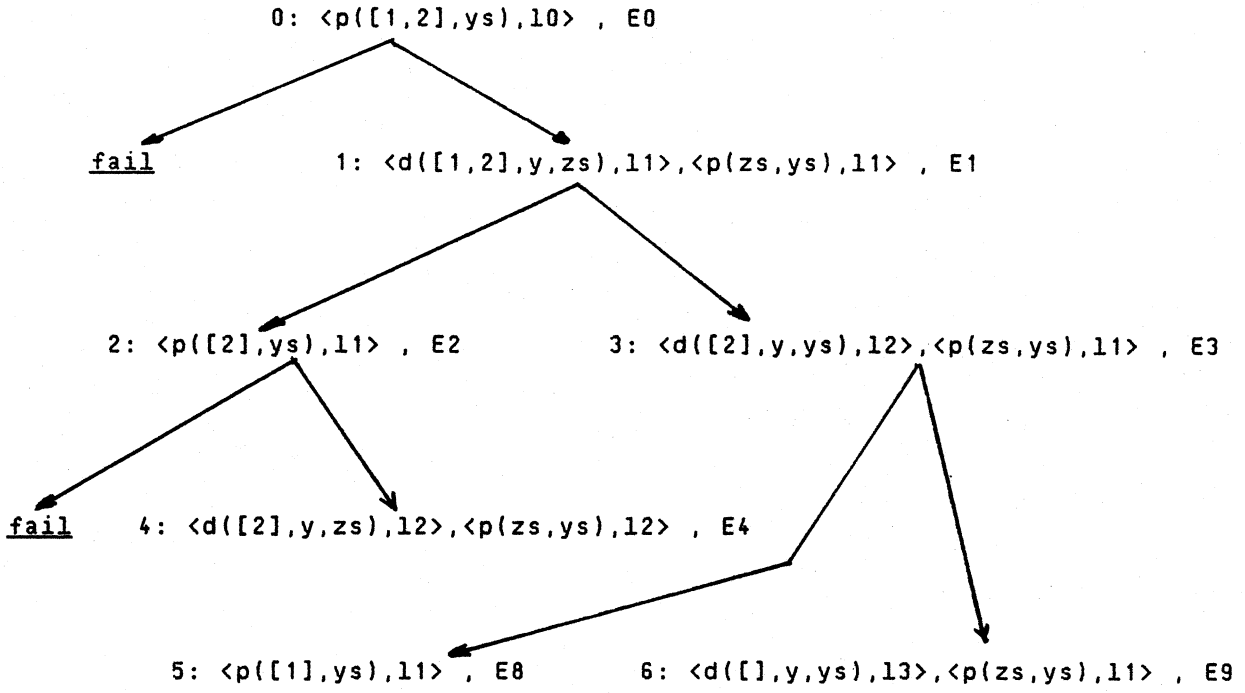
Let us denote the i'th clause of a relation r by r.i, then p.1 and d.1 are assertions, and p.2 and d.2 are implications.

Execution of a program can be described by a search tree. A node in such a tree represents the state of a subcomputation: a sequence of goals and a binding environment:

$$\langle Goal1\rangle,\langle Goal2\rangle,\ldots,\langle Goaln\rangle \ , \ E_i$$

A binding environment consists of contexts containing the values of the variables in the invoked clauses, one context for each clause invocation. Children of a node represent the states reached after executing a goal in the given state.

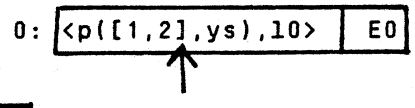
Ex 2: The following figure illustrates the initial four levels of the search tree corresponding to the program in Ex 1. Notice that each goal consists of a literal and a context name l_i which identifies the context containing the values of the variables occuring in the literal. In the figure, the environments E_i are not shown, instead literal substitution of values for variables is used when possible.

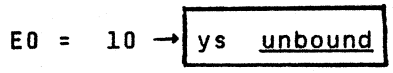


In the interpreter described in [CiHa82A,CiHa83B], a process is created for the root of the search tree. It starts a child process for each clause of the relation chosen by the current goal and then terminates. A newly created process executes unification and if it fails, it terminates, otherwise it creates children processes and then terminates. A branch of computation terminates successfully when there are no more goals to solve. A solution can be extracted from the binding environment.

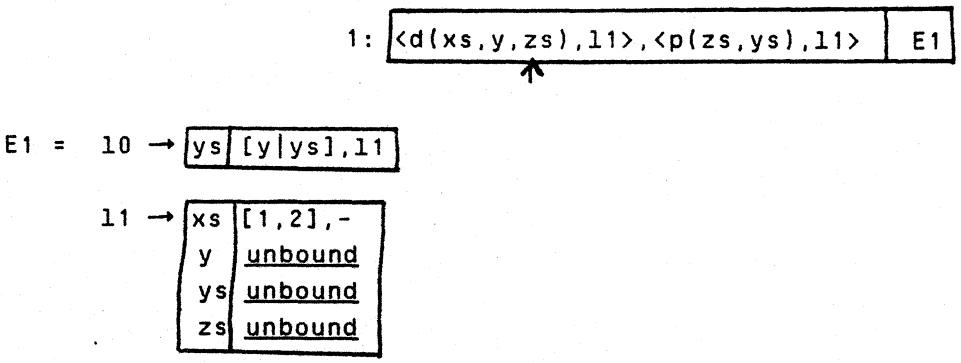
Ex 3: Four snapshots of possible generations of processes for the search tree in Ex 2, where the state of each process is shown. Processes are about to perform a unification step. Current goals are indicated by upward arrows. Environments are also shown in detail where the value of a variable is either unbound or a pair: (Source Term,Context Name).

Snapshot 1: the process corresponding to node 0:

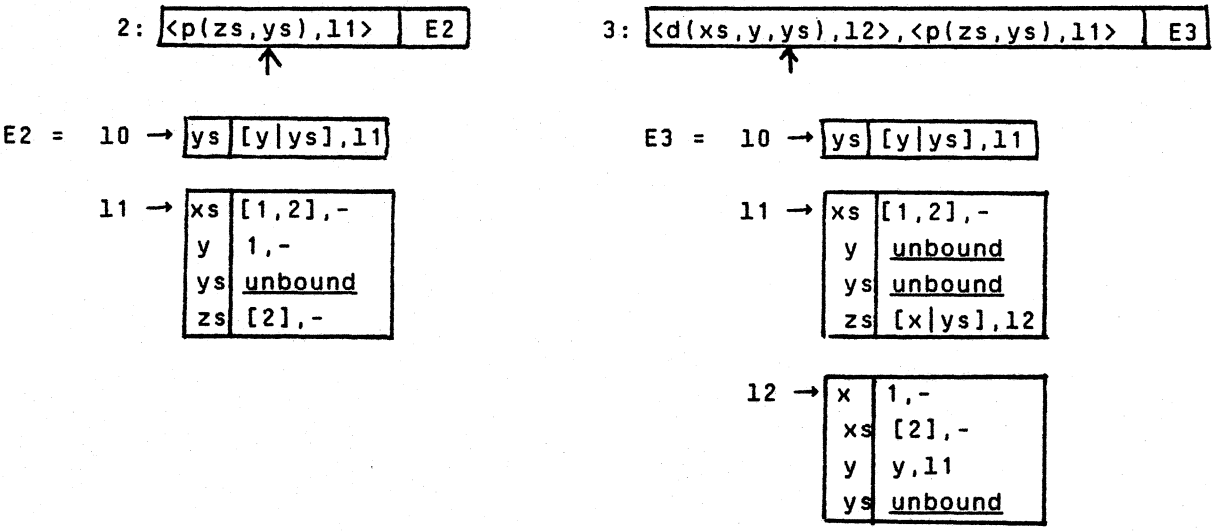




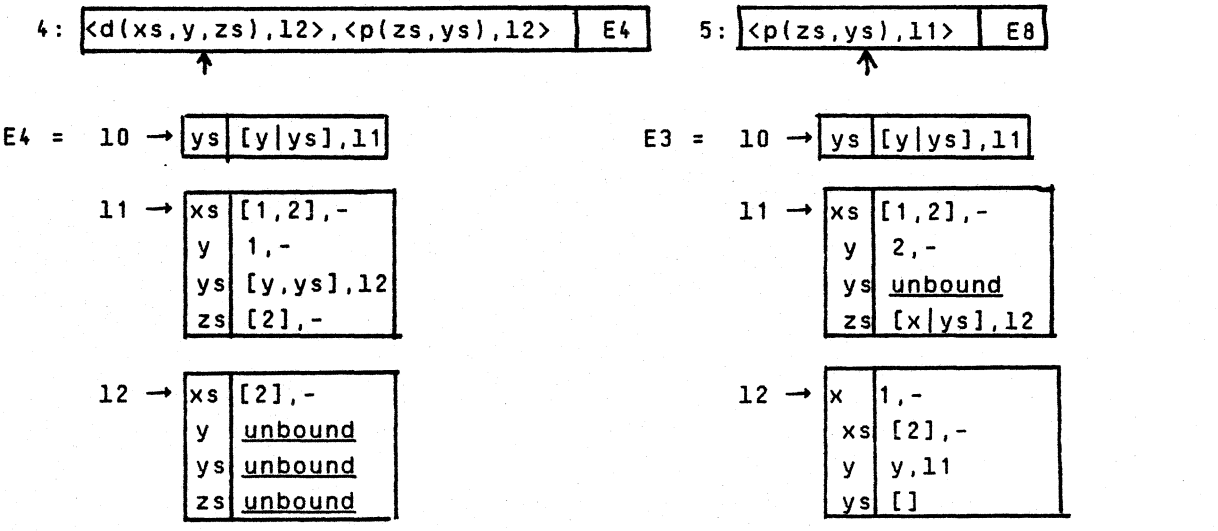
Snapshot 2: the process corresponding to node 1:



Snapshot 3: the processes corresponding to nodes 2 and 3:



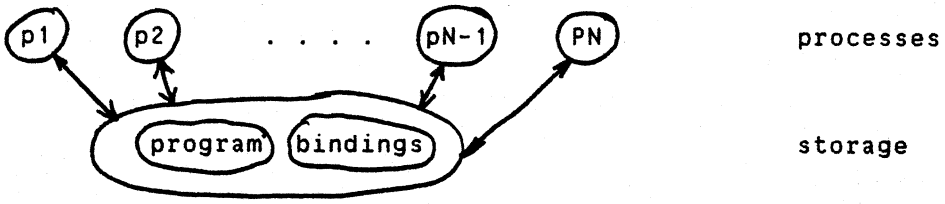
Snapshot 4: the processes corresponding to nodes 4 and 5, we assume that the process corresponding to node 6 and its children have already terminated.



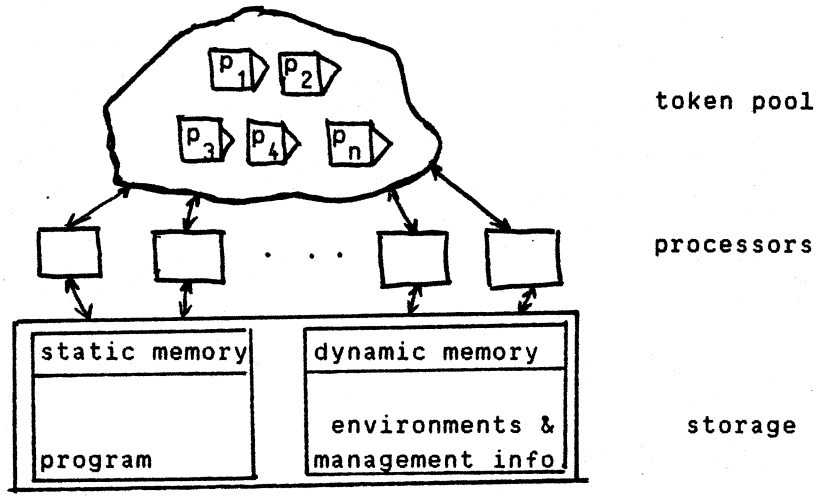
Efficient methods for maintaining a separate address space for each binding environment are described in [CiHa83A,CiHa83B]. For the rest of this paper it

is enough to realise that a variable can be accessed or updated through the unique name: <Environment name,Context name,Variable name>.

A computation described by our earlier interpreter may be visualised as an unlimited number of processes and a storage for binding environments and programs.



In the machine model we present here, the unlimited number of processes is mapped onto a finite number of processors. On this conceptual level we can picture the machine as consisting of a token pool, a set of processors and a storage. Storage is divided into a static memory for programs and dynamic memory for the binding environments and other management information. Tokens in the pool represent processes which are ready for execution but are not allocated a processor. Processors execute processes as prescribed by the tokens and create new tokens. Processors communicate with the storage to access program and data.



The above abstraction is similar to the one presented by Darlington and Reeve in the description of ALICE [DaRe]. It is very useful for handling problems of parallel computations. Furthermore, it can be a starting point for many different architectures.

As mentioned above, the state of a process consists of a list of goals and a binding environment. Such a state will be represented in our machine by a token residing in the token-pool or in one of the processors, and by a possibly empty list of continuation frames residing in the dynamic memory.
A token consists of the following fields:

1. Literal reference (L),
2. Context name (C),
3. Environment name (E),
4. Continuation-Frame reference (CF) and
5. other information to be described later.

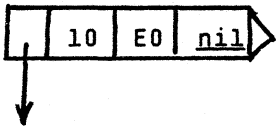
A continuation frame has the following fields:

1. Literal reference (L),
2. Context name (C) and
3. Continuation-Frame reference (CF).

In the next section, when the machine instructions are specified, the L-field in tokens and continuation frames will be a reference to an instruction.

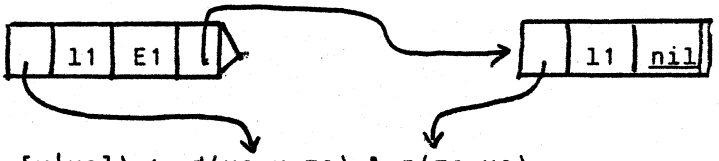
Literals of a clause are selected form left to right. This implies that the head of the goal-list is always the current goal and the tail are the remaining goals. The L and C fields of a token represent the current goal, whereas its continuation frames represent the remaining goals.
These ideas are illustrated by the following snapshots which correspond to those of Ex 3.

Snapshot 1: there is one initial token having the current goal <p([1,2],ys),l0>, and no continuation frame, i.e. field CF is nil.



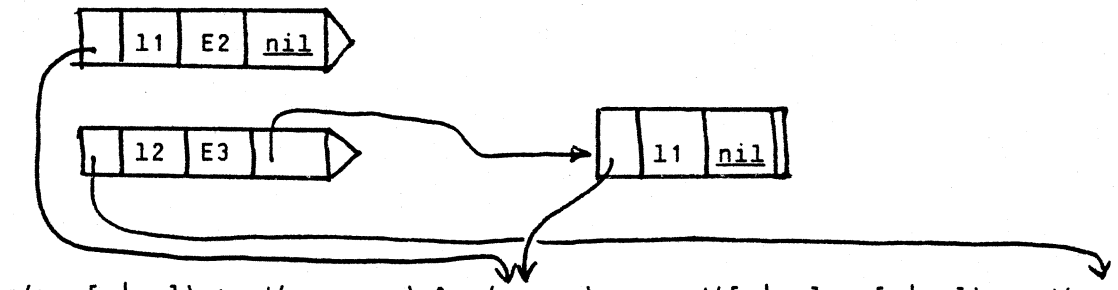
p([1,2],ys)

Snapshot2: one token with the current goal <d(xs,y,zs),l1>, the remaining goal <p(zs,ys),l1> is represented by a continuation frame.



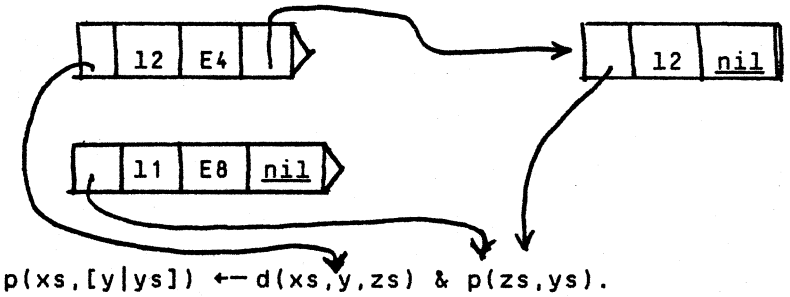
p(xs,[y|ys]) ← d(xs,y,zs) & p(zs,ys).

Snapshot 3: two tokens, the one corresponding to node 3 has a continuation frame.



p(xs,[y|ys]) ←d(xs,y,zs) & p(zs,ys). d([x|xs],y,[x|ys]) ←d(xs,y,ys).

Snapshot 4: evident.



p(xs,[y|ys]) ←d(xs,y,zs) & p(zs,ys).

Continuation frames are read-only data objects. This allows several tokens to share continuation frames.

3. Translation of programs into machine code

In this section, we describe the translation of Horn clauses into machine code. We give also a short summary of the instructions and their effect. The exact specification of the interpretaion cycle of a processor is given in Section 5. Notice that the machine code representation of terms is not given.

We use the following metavariables, which may be indexed, to range over basic syntactic entities:

Terms:	t,q,r,s.
Relation names:	R,S.

By ↑t and ↑R we mean a reference to the representation of the term t and to the relation (clause) R respectively.

An assertion having m variables:

$$R(t_1,\ldots,t_n)$$

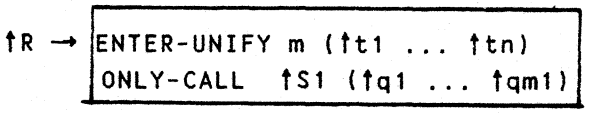
is translated into:

```
↑R → ENTER-UNIFY m (↑t1 ↑t2 ... ↑tn)
     RETURN
```

An implication with only one literal in its body and having m variables:

$R(t_1,\ldots,t_n) \leftarrow$
$\quad S_1(q_1,\ldots,q_{m1})$.

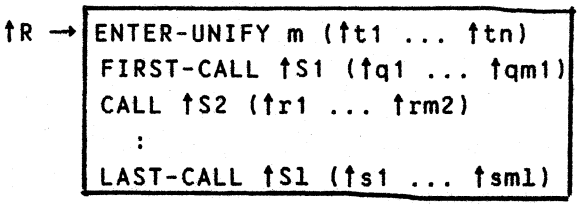
is translated into:



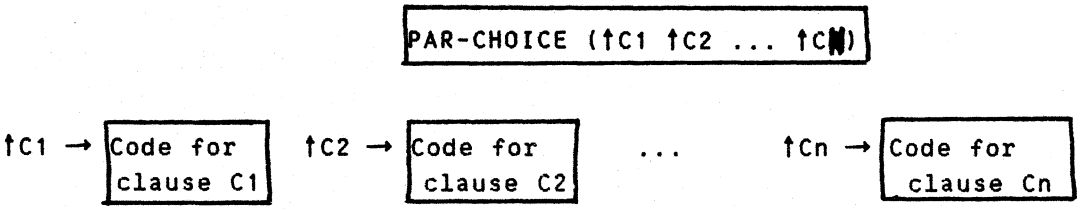
An implication with two or more literals in its body and having m variables:

$R(t_1,\ldots,t_n) \leftarrow$
$\quad S_1(q_1,\ldots,q_{m1})$ &
$\quad S_2(r_1,\ldots,r_{m2})$ &
$\quad\quad\vdots$
$\quad S_l(s_1,\ldots,s_{ml})$

is translated into:



A Relation R consisting of several clauses, C1 C2 ... Cn, is translated into:



A processor fetches a token and executes the instruction it refers to. After the instruction is performed the processor may create none, one or more tokens according to the interpreted instruction. As mentioned earlier every token has a list of continuation frames. The description that follows, of the instructions, will be relative to the token being interpreted, so "Remove first continuation frame" actually means to remove the first continuation frames from the list associated with the interpreted token.

(1) ENTER-UNIFY m (↑t1 ↑t2 ... ↑tn) :
Create a variable-context for m variables in current environment. Execute a unification step; the callers parameters are referred to in the interpreted token.

(3) RETURN :
Return control to the caller. The next instruction to be executed is stored in the first continuation frame.

(4) ONLY-CALL ↑S (↑t1 ... ↑tn) :
Transfer control and parameters to S; this instruction is used where there is exactly one literal in the body of a clause and therefore no continuation frames are created.

(5) FIRST-CALL ↑S (↑t1 ... ↑tn) :
Create a continuation frame; save next instruction in it and link it first in the continuation frame list; transfer control and parameters to S.

(6) CALL ↑S (↑t1 ... ↑tn) :
Remove the first continuation frame; link first a continuation frame referring to next instruction; transfer control to S.

(7) LAST-CALL ↑S (↑t1 ... ↑tn) :
Remove the first continuation frame; transfer control to S.

(8) PAR-CHOICE (↑C1 ↑C2 ... ↑Cn) :
Create n tokens each having its own environment; i.e. n parallel activities are initiated. The created tokens share the continuation frame list of the interpreted token.

4. Notational Conventions

The next section specifies the interpretation cycle of a processor. We present here the essential characteristics of the specification language used there. The language used may be considered as an imperative fraction of Meta-IV [BjJo].

4.1. Types of Objects

The elementary type NAT is the class of all natural numbers 0, 1, ... and the type BOOL is the class of truth values true and false. Elementary types like unbound is meant to be the singleton set with the element unbound.

Lists

The type of lists of objects each having the type A is denoted by
A*
The list of the objects e1, e2, ... ,en in this order is formed using
<e1,e2, ...,en>
An empty list is denoted by nil. The following operations apply to a list l where l = <e1,e2, ...,en>:
l[i] == ei (yields the i'th element of a list),
len l == n (the length of the list l)

Reference types

Let A be a type then

↑A

is the type of references (addresses) of objects of type A. If i has the type ↑A, then the operation i↑ returns the object a of type A referred by i otherwise it returns nil.

Cartesian products

The type of heterogenous n-tuples for which the first object is of type A1, the second object is of type A2, etc. is denoted by

A1 A2 ... An

An object of this type is treated as a list of length n.

Abstract Types

Abstract types of compound objects may be specified by means of the following rules:

(1) A = B1 | B2 | ... | Bn

This rule defines the abstract type named A (a type identifier) to be the union of the (disjoint) types defined by B1, B2, ..., Bn, where Bi are type identifiers or type expressions as defined above.

(2) A :: B1 B2 ... Bn

This rule defines the type A to be the type of A-tagged n-tuples of the type (B1 B2 ... Bn). An A-tagged n-tuple object is formed with the expression

mk-A(e1,e2,...,en)

where 'mk-' is the so called make constructor. The above expression generates the tuple <e1,e2,...,en> equipped with the tag 'A'.

(3) A = B1 B2 ... Bn

The same as rule (2) however tuples are not tagged.

4.2. Statements

A crucial statement used in the specification below is

```
(def x: e;
 S
)
```

where x is an identifier, e is an expression possibly having some side effect (e.g. a procedure returning a value), and S is a statement. The expression e is evaluated first, then all occurrences of x in S are replaced by the value returned by e, finally S is evaluated in this context. More generally in the construct

```
(def mk-A(x1,x2,...xn) = e;
 S
)
```

e is evaluated to yield an A-tagged n-tuple, the immediate components of which are then denoted by x1, x2, ..., xn in the evaluation of the statement S.

The other forms of statements are familiar from other imperative languages (Pascal etc.). For example sequential statement composition has the form:

(S1;S2; ...;Sn),

cases have the form:

cases e0 : (e1 → S1, e2 → S2, ... , en → Sn)

and the indexed iteration:

for i = m to n do S(i)

A definition of a procedure F returning a value in our specification language is assigned a type of the form:

F: B1 B2 ... Bn => B (n ≥ 0)

telling that F has n arguments that are of the types B1, B2, ..., Bn and returns a value of the type B.

If F does not return a value, i.e. is applied for its side effect only, F will get a type of the form

F: B1 B2 ... Bn =>

5. Specification of the processor cycle

The instructions introduced in the previous sections are executed by each processor. Here, we define the basic execution cycle of the processor and the exact meaning of the instructions.

5.1. Instruction set

A program consists of the initial call and a sequence of instructions.

```
Program = INIT-CALL Code
Code = Instruction*
```

There are following instructions:

```
Instruction = INIT-CALL | FIRST-CALL | CALL | LAST-CALL | ONLY-CALL |
              PAR-CHOICE | ENTER-UNIFY | RETURN
```

With the following syntax:

```
INIT-CALL :: ↑Instruction Nat (↑Parameter)*
FIRST-CALL :: ↑Instruction (↑Parameter)*
CALL :: ↑Instruction (↑Parameter)*
LAST-CALL :: ↑Instruction (↑Parameter)*
ONLY-CALL :: ↑Instruction (↑Parameter)*
PAR-CHOICE :: (↑Instruction)*
ENTER-UNIFY :: Nat (↑Parameter)*
RETURN :: nil
```

Both Parameters and instructions are stored in the static memory.

5.2. Tokens and continuation frames

The state of a process is represented by a token and a list of read-only continuation frames stored in the dynamic memory. Here follows the definition of a token (Token) and a continuation frame (Cont-Frame) which were schematically introduced in Section 2.

```
Token :: ↑Instruction Context-Name ↑Environment  ↑Cont-Frame (↑Parameter)*
Cont-Frame :: ↑Instruction Context-Name ↑Cont-Fram
```

↑Environment refers to the process's environment directory, and Context-Name is used to lookup the designated context in this directory. (↑Parameter)* is the list of parameters in a call instruction.

5.3. Execution cycle

In each cycle a processor fetches a token from the token pool, fetches the referred intruction from the static storage, and finally decodes and executes the instruction. A result of an instruction is none, one or more tokens. No more tokens means that this branch of the search tree has terminated, either with success or with failure. One token means that the current branch is continued. More tokens means that a nondeterministic point has been encountered and a fork into new branches has occurred.

The interpretation cycle of a processor is shown below. A number of auxiliary functions, or procedures, are used there. These functions are divided mainly into two groups: (1) functions operating on the token pool, and (2) functions operating on the dynamic storage for managing environments and variable-contexts.

Token management

```
FetchToken : => Token
    Delivers a token form the token pool to the calling processor.
SendToken : Token =>
    Sends a token to the token pool.
```

Binding environment management

The following operations are described in detail in [CiHa83A,CiHa83B]:

DuplicateEnv : ↑Environment => ↑Environment

Creates a logical copy of the input environment and returns a reference to the newly created environment.

ReleaseEnv : ↑Environment =>

Reclaims the storage of the input environment and all its contexts that are no longer accessible.

SendSolution : ↑Environment =>

SendSolution(e) extracts the bindings of the variables in the first context in e and then performs a ReleaseEnv operation.

NewContext : ↑Environment Nat => Context-Name

NewContext(e,n) creates a new context of n variables in e and returns its name.

Unify: (↑parameter)* Context-Name (↑parameter)* Context-Name ↑Environment => BOOL

Unify executes of the unification algorithm, it accesses and assigns values of variables.

One more auxiliary function is

NextInstr : ↑Instruction => ↑Instruction

NextInstr(i) returns a reference to the instruction following i↑.

Here follows the processor cycle.

```
Instruction-Processor processor() ≜
    (cycle
        (def mk-Token(i,c,e,cf,ps): FetchToken();
         cases i↑ :

            mk-FIRST-CALL(i1,ps1) →
                (def cf1 : New(mk-Cont-Frame(NextInstr(i),c,cf));
                 SendToken(mk-Token(i1,c,e,cf1,ps1))
                ),
            mk-CALL(i1,ps1) →
                (def mk-Cont-Frame( , ,cf1) : cf↑;
                 def cf2 : New(mk-Cont-Frame(NextInstr(i),c,cf1));
                 SendToken(mk-Token(i1,c,e,cf2,ps1))
                ),
            mk-LAST-CALL(i1,ps1) →
                (def mk-ContFrame( , ,cf1) : cf↑;
                 SendToken(mk-Token(i1,c,e,cf1,ps1))
                ),
            mk-ONLY-CALL(i1,ps1) →
                SendToken(mk-Token(i1,c,e,cf,ps1)),
            mk-PAR-CHOICE(is) →
                (for i=1 to len is do
                   SendToken(mk-Token(is[i],c,DuplicateEnv(e),cf,ps));
                 ReleaseEnv(e)
                ),
            mk-ENTER-UNIFY(n,ps1) →
                (def c1: NewContext(e,n);
                 if Unify(ps,c,ps1,c1,e) then
                    SendToken(mk-Token(NextInstr(i),c1,e,cf,nil))
                 else    !Failure
                    ReleaseEnv(e)
                ),
            mk-RETURN() →
                (if cf=nil then     !Success
                    SendSolution(e)
                 else
                    (def mk-Cont-Frame(i1,c1,cf1) : cf↑;
                     SendToken(mk-Token(i1,c1,e,cf,nil)))
                    )
                )
        )
    )
```

6. Discussion

During an Or-parallel execution the number of processes, as prescribed by their tokens, usually exceeds the number of available processors. The problem of storing the state information during the traversal of a search tree is not special for our parallel machine. In a sequential machine, information about not yet executed alternatives must be saved. Breadth-first traversal of the search tree usually leads to a combinatorial explosion of the space requirement. Therefore, practical logic programming systems control the traversal of the search tree usually by using a depth-first traversal strategy combined with a mechanism for pruning some branches of the search tree. Such a mechanism takes the form of a rudimenary Cut (Slash) operator as in Prolog, or intelligent backtracking or both.
Similarly, any feasible parallel machine should incorporate mechanisms for
(1) controlling the traversal of the search tree, and
(2) pruning some branches of the search tree.

The first issue can be reduced to that of adopting a proper policy for scheduling the tokens on the available processors. For instance, if the token pool has the form of a LIFO queue, and each processor keeps always one of the tokens it produces and sends the other tokens to the queue, our parallel machine would then work as a 'broad' depth-first machine, investigating in parallel a number of branches that is equal to the number of processors. That is to say, having n processors we get approximately n Prolog machines working in papallel. The centralised access of such a token pool would presumably create a bottle-neck in the system and have to be approximated by partitioning the token pool on the processors. This issue will be treated in a forthcoming paper.

The second issue requires either an extension to the source language, or a seperate control language. Very often one would like to get exactly one solution for a subgoal. This happens, for example, when the relation defined is in fact a function for certain patterns of arguments, or when it is a test predicate with all arguments instantiated, or for several other reasons. Once a solution for a goal is found, the other branches in the search tree having the same goal can be pruned. Translating this into our machine means that a mechanism for aborting certain tokens should be available. Such a mechanism does not require any process hierachy nor message passing between processes. The extended machine incorporating a mechanism solving this, and other problems, is described in another paper [CiHa83C].

Our abstract machine can, be classified as an unconventional control-driven machine [Tr]. The execution sequence is decided by the flow of control in tokens. It is interesting to compare it with a data-driven abstract machine proposed by Umeyama and Tamura[UmTa]. In their proposal a program is represented by a dataflow graph. Tokens carry instantiated goals, subtitution sets or both. Tokens are dynamically tagged to distinguish different invocations of the same clause. Tokens with the same tag must be matched during the execution. We consider the dataflow principle is an unnecessary complication

in an Or-parallel machine, because of the overheads in both creating unique tags and matching tokens with the same tag. Dataflow is not needed because the control flow of the programs can be determined at compile time regardless of the arrival of data. The dataflow principle might be interesting when some form of and-parallelism is considered.

References

[BjJo] D Bjørner, C B Jones (eds), The VDM: The Meta Language, Lecture Notes in Computer Science 61, Springer-Verlag 1980

[CiHa83A] A Ciepielewski, S Haridi, Storage Models for Or-parallel execution of Logic Programs, TRITA-CS-8301, Royal Institute of Technology, Stockholm 83

[CiHa83B] A Ciepielewski, S Haridi, A Formal Model for Or-parallel execution of Logic Programs, to appear in IFIP 83, North Holland P. C., Mason (ed)

[CiHa83C] A Ciepielewski, S Haridi, Control of Activities in an Or-parallel Token Machine, in this proceedings.

[CoKi] J S Conery, D F Kibler, Parallel Interpretation of Logic Programs, ACM Symposium on Functional Programming Languages and Computer Architecture, October 1981

[DaRe] J Darlington, M Reeve, ALICE: A Multiprocessor Reduction Machine for the Parallel Evaluation of Applicative Languages. Proceeding of ACM Conference on Functional Programming Languages and Computer Architecture, 1981.

[EKM] N Eisinger, B Kasit, J Minker, Logic Programming a Parallel Approach, First Logic Programming Conf., Marseille 1982

[FNM] K Furukawa, K Nitta, Y Matsumoto, Prolog Interpreter Based on Concurrent Programming, in proceedings of the First International Logic Programming Conference, Marseille 82

[HaSa] S Haridi, D Sahlin, An Abstract Machine for LPL0 TRITA-CS-8302, Royal Institute of Technology, Stockholm 83

[Po] G H Pollard, Parallel Execution of Horn Clause Programs, PhD Thesis, Imperial College of Science and Technology, University of London, 1981

[Tr] P C Treleaven, D R Brownbridge, R P Hopkins, Data-Driven and Demand-Driven Computer Architecture, ACM Computing Surveys, vol 14, No 1, March 1982

[RoSi] J A Robinson, ε E Siebert, LOGLISP: Motivation, Design and Implementation, in Logic Programming edited by K L Clark and S-Å Tärnlund, Academic Press 82

[UmTa] S Umeyama, K Tamura, Parallel Execution of Logic Programs, in proceedings of the 10 Symp. on Computer Architecture, Stockholm 83

AN EXPERIMENT IN AUTOMATIC SYNTHESIS OF EXPERT KNOWLEDGE THROUGH QUALITATIVE MODELLING

I.Mozetič (1), I.Bratko (1,2), N.Lavrač (1)

(1) Institute Jozef Stefan, Jamova 39, Ljubljana, Yugoslavia
(2) Faculty of Electrical Engineering, Trzaska 25, Ljubljana

Abstract

We have developed a qualitative model of the heart for the simulation of its electrical behaviour. The model was used to automatically generate a knowledge-base of all physiologically possible combinations of cardiac arrhythmias and their corresponding ECG descriptions. The knowledge thus generated was verified by cardiologists and is used by a medical expert system. The model of the heart is formally expressed in a subset of the first-order logic. The qualitative simulation is carried out by a simple and efficient inference mechanism implemented in Prolog.

Introduction

We have developed the diagnostic part of an expert system for the diagnosis and treatment of patients with cardiac arrhythmias to be used at the University Medical Centre in Ljubljana. In the paper we concentrate on the ECG interpretation module which now includes a qualitative model of the heart. There were at least four reasons for deepening the system's knowledge by including the model. The physiological knowledge about the heart is of great importance for:

- finding the causes of arrhythmias,
- for choosing an appropriate treatment of diseases,
- for intelligent explanation of the system's answers,
- for automatically generating the electrocardiographic knowledge-base for the combinations of single arrhythmias already known to the system.

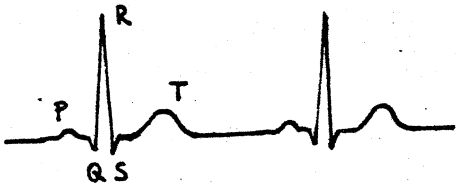
This last reason was in fact our immediate goal.

Our model is qualitative and developed along similar lines as e.g. the work of Forbus (1982) or de Kleer (1977). One reason why a qualitative model is a natural choice is that the physiological descriptions of the heart are largely qualitative. Another reason is that for a computer simulation based on a quantitative model, numerical values of the model parameters for a given patient would be needed. Such parametrs, however, practically cannot be measured. A similar aproach to medical diagnosis is exemplified in the CASNET system (Weiss, Kulikowski, Amarel, 1978).

Interpretation of ECG

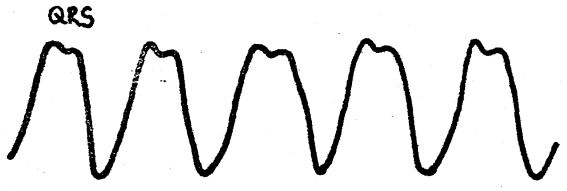
Fig. 1 shows two ECG diagrams, the first for a normal heart, and the second for ventricular tachicardia, one of the arrhythmias that are handled by the system. The ECG is in the system represented by its qualitative description rather than by an actual voltage vs. time relationship. The description of a given ECG diagram consists of elementary patterns present in the ECG diagram and the relations between these patterns.

The medical literature on the relationship between various heart disorders and their corresponding ECG diagrams (e.g. Phibbs 1973, Mandel 1980) is quite indicative of the nature of these elementary patterns. However, we could not find any definite proposal, or formalisation, of a complete and compact set of such patterns. The language that we designed for describing ECG consists of a set of 10 attributes, each of which having typically 3 or 4 values. Fig. 1 shows two examples of such descriptions.



normal sinus rhythm

rhythm: regular,
frequency: between_60_100,
frequency_P: between_60_100,
regular_P: normal,
relation_P_QRS: after_P_QRS,
regular_PR: normal,
regular_QRS: normal



ventricular tachycardia

rhythm: regular,
frequency: between_100_250,
regular_P: absent,
regular_QRS: wide

Fig. 1: Two ECG diagrams and their qualitative descriptions.

The construction of a knowledge-base which covers the relation between 26 simple cardiac arrhythmias and their corresponding ECG diagrams was relatively straightforward. It was completed in consultation with cardiologists in about three months. The relation between the arrhythmias and ECG is in the system represented by rules of the form:

if diagnosis then ECG-description

For example:

if ventricular tachicardia
then rhythm is regular and
frequency is between 100 and 250 and ...

Accordingly, this part of the knowledge-base is used not to confirm some diagnosis, but to eliminate those diagnoses that contradict the patient's ECG. The remaining set of non-eliminated diagnoses (typically a few diagnoses) is then input for the differential diagnosis in which clinical data is used. The clinical knowledge ranks the remaining arrhythmias by estimating their relative likeliness.

This knowledge-base is, however, not sufficient for dealing with the more difficult problem of diagnosing the patients with multiple arrhythmias. As the number of combinatorially possible multiple arrhythmias (combined of 2, 3, 4 etc. single ones) exceeds hundred thousand, the direct specification of their ECG descriptions by exhaustive manual tabulation is practically impossible. Also, there is no systematic and exhaustive treatment of multiple arrhythmias in the medical literature.

This conclusion motivated, among other reasons, the development of a model of the heart to facilitate the automatic derivation of the relation between multiple heart failures and their corresponding ECG descriptions. With the introduction of the model, the knowledge base was "deepened" as illustrated in Fig. 2.

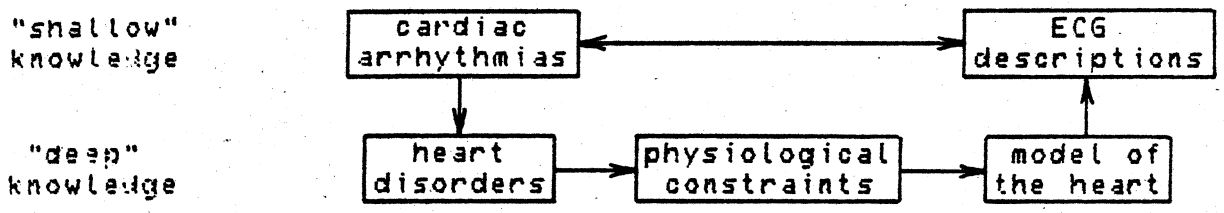


Fig. 2: "Shallow" and "deep" diagnostic knowledge. In the deep knowledge diagnoses are defined in terms of heart disorders; if a set of disorders is physiologically possible it instantiates the model of the heart; the corresponding ECG is derived by running the model.

The model of the heart

For its electrical behaviour, the heart can be represented as a network consisting of: impulse generators, impulse propagation paths and summation elements for impulses as shown in Fig. 3. In the medical literature we can find the following definition of the cardiac arrhythmias: The cardiac rhythm - be it normal or abnormal - can be characterized and classified with respect to the characteristics of impulse origin, discharge sequence and impulse conduction (WHO/ISFC Task Force 1979). These characteristics are of following types: A generator can be silent, or an extra (ectopic) generator may appear; impulse propagation paths can be partially or totally blocked, or extra paths may apoear, ...

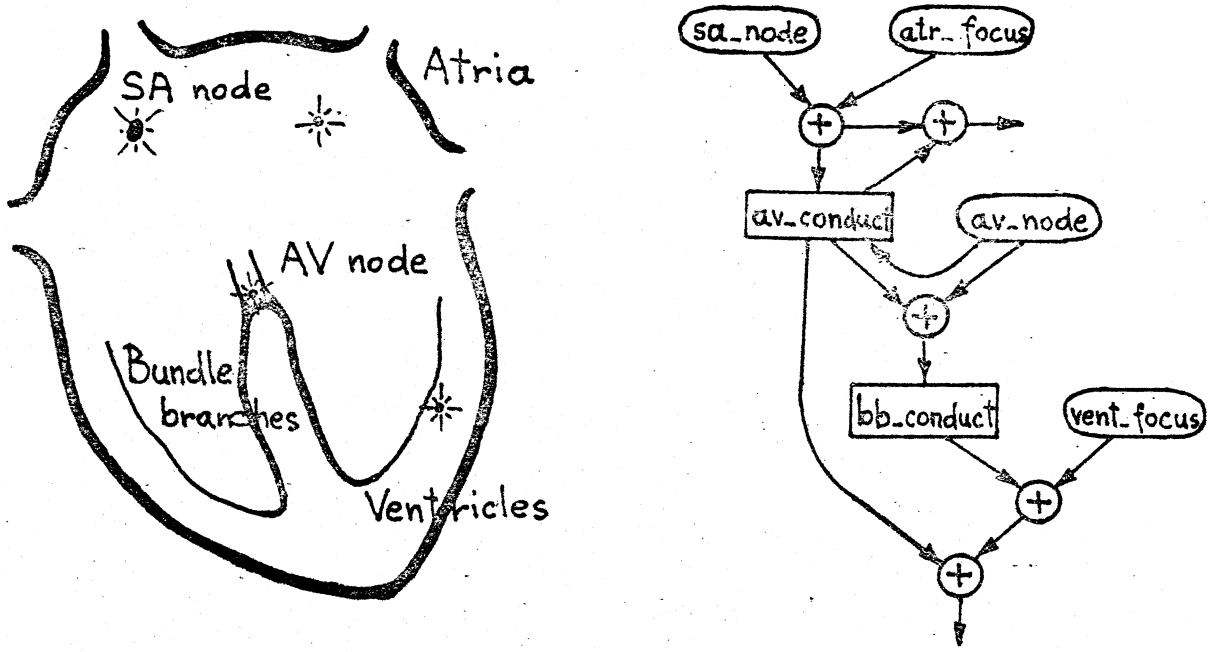


Fig. 3: A scheme of the heart and the overall logic of the model.

The state of the heart is represented by the states of its parts (4 impulse generators, 2 propagation paths and the rate of atria and ventricles). Each simple arrhythmia is defined as the abnormal state of one of the heart parts; other parts are assumed to be normal. Two or more arrhythmias can be combined if they do not contradict (e.g. are not defined by different states of the same heart part).

There are additional physiological constraints on the state of the heart. The constraints in the model are based on an assumption that malfunctions of impulse generators, giving permanent rhythm can be mutually combined only if there is a complete conduction block between them. Even if these malfunctions are sometimes physiologically possible, they cannot be seen on the surface ECG leads and are never considered by physicians.

The model defines relations between parts of the heart, electrical impulses and corresponding ECG descriptions. Formally, it is expressed as a set of if-then rules in a clausal form of the first-order logic. It was possible to order the list of rules according to the following principle: For each pair of rules R1 and R2, R2 may proceed R1 only if no literal in the consequent of R2 occurs in the antecedent of R1. This implies that there is no cyclic or recursive rules. This constraint on the list of rules facilitates fast, one pass execution of the model.

The inference mechanism that runs the model for a given multiple arrhythmia is relativly simple. The model of the heart is first instantiated with the state of the heart parts. The states of the heart parts are added to the set of rules as unit

clauses. Then the inference mechanism sequentially passes through the rules and by applying modus ponens derives all positive facts. One pass through the ordered list of rules suffices for generating all possible ECG descriptions which correspond to this multiple cardiac arrhythmia.

Rules, 62 of them, define relations of the following types:

1. generator → impulse

2. impulse → propagation path → impulse

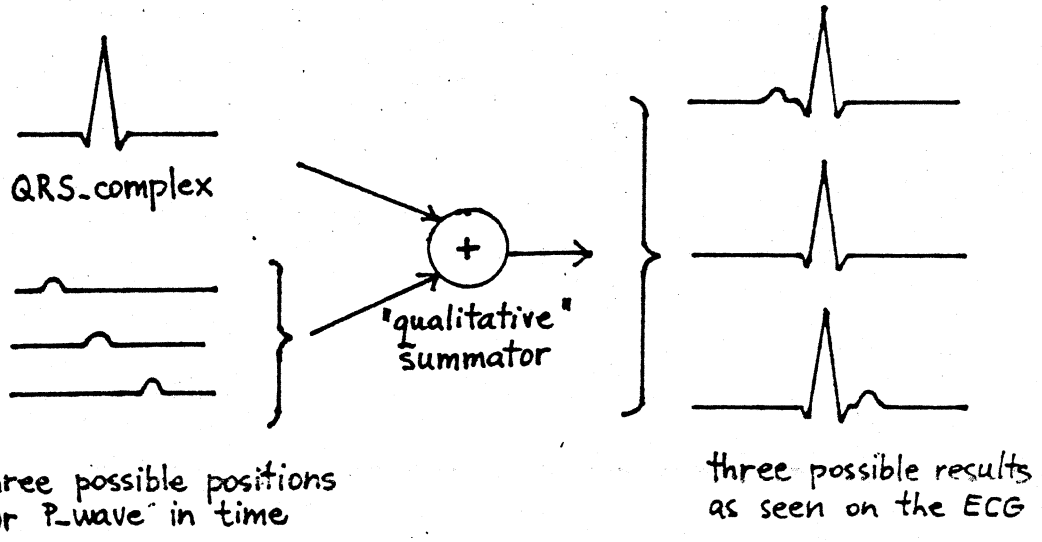
3. impulse, impulse → + → impulse

4. impulse, impulse, ... → ECG-description

An example of a rule is:

<u>if</u> there are ectopic impulses at the His bundle and in the supraventricles originating at the AV focus
<u>then</u> this results in the following ECG features: either a short PR interval, or no P wave, or P wave after the QRS complex

These three cases result from the "qualitative summation" (as also percieved in the ECG diagram) of two signals which can be relatively shifted in time in three ways as shown in Fig. 4.



<u>Fig. 4:</u> The qualitative summation of ECG patterns.

Implementation and results

We ran the system for all combinations of simple cardiac arrhythmias. A large proportion of the corresponding states of the heart parts were recognized as physiologically impossible. For the physiologically possible arrhythmias the model generated corresponding ECG descriptions. The following table shows the number of mathematically and physiologically possible arrhythmias against the number of their constituent single arrhythmias.

No. of constituent arrhythmias	1	2	3	4	5	6	7
No. of mathematical combinations	23+3	253	1771	8855	33649	100947	...
No. of physiologically possible combinations	13+3	85	231	163	73	20	0

Note that some arrhythmias cannot occur alone (e.g. blocks), but only in combination with others (e.g. sinus rhythm). Three arrhythmias cannot be combined with others.

The whole system is implemented in Prolog on DEC-10 (Pereira, Pereira, Warren 1978). The compiled program generated ECG descriptions for all combinations of arrhythmias in 340 CPU seconds.

The thus obtained knowledge-base of ECG descriptions for all possible multiple arrhythmias can be used for diagnosis. If an explanation is requested then for a given ECG description the corresponding states of the heart parts are retrieved by table look-up. The model is then re-run for these states and its trace can serve as an explanation. We have not yet found an efficient implementation of the model to be run in the inverse direction, i.e. from the ECG toward the diagnoses.

Conclusion

The model facilitated, as our main result, the automatic derivation of an exhaustive catalog of multiple arrhythmias and their corresponding qualitative ECG descriptions.

The present system handles 588 combinations of arrhythmias, all of them physiologically possible and majority of them can be observed in everyday clinical praxis. The importance of recognition of these arrhythmias is very different. Sometimes the diagnosis of certain rhythm disturbances is critical for the treatment. On the other hand, some arrhythmias are only of

theoretical interest without practical consequences for the patient.

The model of the heart also provides a good basis for the system's explanation of its own reasoning, and (as hoped) for the treatment decision-making. We are planning to extend the model in two directions:

- to handle the mechanical activity of the heart as well as electrical;
- to provide causal reasoning about the effects of drugs and their interaction for treatment decision-making.

Acknowledgement

The authors would like to thank cardiologists prof. M. Horvat, P. Rode, B. Čerček and A. Grad of the University Medical Centre in Ljubljana for the consultations in the development of the knowledge-base.

References

[1] de Kleer J. (1977) Multiple representations of knowledge in a mechanics problem-solver. Proceedings of IJCAI 1977.

[2] Forbus K.D. (1982) Qualitative Process Theory. MIT, AI Lab, AI Memo 664.

[3] Mandel W.J. (1980) Cardiac Arrhythmias. J.B. Lippincott Co.

[4] Pereira F., Pereira L.M, Warren D.H.D. (1978) User's Guide to DEC-10 Prolog. University of Edinburgh, Dept. of AI.

[5] Phibbs B. (1973) The Cardiac Arrhythmias. The C.V. Mosby Co.

[6] Weiss S.M., Kulikowski C.A., Amarel S. (1978) A model-based method for computer-aided medical decision-making. Artificial Intelligence, Vol. 11, pp. 145-172.

[7] WHO/ISFC Task Force, Classification of cardiac arrhythmias and conduction disturbances. American Heart Journal, Vol. 98, No. 2, Aug. 1979.

EVALUATION OF LOGIC PROGRAMS BASED ON NATURAL DEDUCTION

Seif Haridi and Dan Sahlin
Department of Telecommunication and Computer Systems
Royal Institute of Technology
Stockholm, Sweden

(D R A F T)

0. Introduction

In this paper we show how a large subset of first order logic can be reasonably efficienty interpreted. Logic programming has usually been restricted to a conditional type of statements called Horn clauses. General logical statements that are natural to write cannot directly be expressed by Horn clauses. This is due to the fact that Horn clauses express only the "if-halves" of "iff-definitions". This has meant that this that are easily expressed in first order logic has been done in meta-logic. For example, negation has been treated as nonprovability and special 'setof' contructs have been devised to find all the solutions to a relation.
To solve these problems we construct an abstract machine called 'gepr' (=goal, environment, program and resumption register). The ideas of the 'gepr' machine resemble the 'secd' machine for functional programming languages [La63][He80], which describes what state transitions that are allowed. Although we have used a version of Horn clauses to describe the state transitions, they could equally well have been described in an imperative language.
The basis for the interpreter are the rules of a natural deduction system as shown in [Ha81].

1. Sample programs

A program consists of a set of relation definitions, where a relation is a predicate-logic statement of one of the forms:

1. relation_name($term_1$,$term_2$,...,$term_n$) <-> arbitrary logic statement

2. relation_name($term_1$,$term_2$,...,$term_n$) -> arbitrary logic statement

3. relation_name($term_1$,$term_2$,...,$term_n$) <- arbitrary logic statement

4. ¬ relation_name($term_1$,$term_2$,...,$term_n$)

We also apply a rule of implicit quantification for the variables not being quantified:

> Variables occuring in the "head" of the relation are universally quantified over the whole statement, while the other variables are existentially quantified over the right hand side of the relation definition.

For example
```
     list(w) <-> w=[] or w=[x|y] & list(y)
```
actually means in logic
```
     ∀w( list(w) <-> ∃x∃y (w=[] or w=[x|y] & list(y)) )
```

To be able to interpret the statement above, we break it down further into the conjunction of two statements:

```
     ∀w( list(w) -> ∃x∃y (w=[] or w=[x|y] & list(y)) )
```
and
```
     ∀w( list(w) <- ∃x∃y (w=[] or w=[x|y] & list(y)) )
```

The fourth type of statement given above, the negation, is also transformed into an implication. For example,
```
     ¬ member(x,[]).
```
is transformed to
```
     member(x,[]) -> False.
```

Some more examples:

The full definition set of 'member':

```
¬ member(x,[]).
member(x,[y|z]) <-> x=y or member(x,z).
```

The predicate 'class' tests if all members of 'sl' take course c, or for a certain course finds all its members etc.

```
class(c,sl) <-> ∀s (takes_course(s,c) <-> member(s,sl)).

takes_course(x,y) <-> x=D & y=C1 or
                      x=J & y=C1 or
                      x=J & y=C3.

maths_course(z) <-> z=C1 or z=C3.
```

A "maths major" is a person who takes all maths courses:

```
maths_major(x) <-> ∀y (maths_course(y) -> takes(x,y)).
```

This enables us to find out that maths_major(J) is true, or even makes it possible to find all "maths majors" with a variant of the 'class' predicate above.

2. Types

To simplify the description of the interpreter below, we here introduce a type concept in first order logic. What we actually do is that we have a convenient way to define relations that are true iff their arguments are in a certain domain. We will not here try to make an exact definition of the transformation between our simplified notation and logic, but just show a few examples.

Having the type definition

```
TYPEDEF formula = And(formula,formula) ! Or(formula,formula) ! Eq(term,term)
```

we get the corresponding logic statement

```
formula(x) <-> ∃y∃z(x=And(y,z) & formula(y) & formula(z) ) or
               ∃y∃z(x=Or(y,z)  & formula(y) & formula(z) ) or
               ∃y∃z(x=Eq(y,z)  & term(y) & term(z)).
```

To be able to define a list of a certain type

```
TYPEDEF list(t) = [] ! [t|list(t)]
```

we find it convenient to use schemas in first order logic:

```
list(t)(x) <-> x=[] or ∃y∃z( x=[y|z] & t(y) & list(t)(z) ).
```

(A schema can take a relation as an argument).
Obvious abbreviations are also used in the type definitions.

3. Formulas

The full type definition of a formula looks like this

```
TYPEDEF formula = And(formula,formula) ! Or(formula,formula) !
                  Imp(formula,formula) ! False !
                  Eq(term,term) ! Rel(name,list(term)) !
                  All(list(name),formula) ! Exist(list(name),formula).

TYPEDEF term = Var(name) ! Dstruct(name,list(term))
```

and the relation 'name' is appropriately defined.

For example, the formula "∀y (maths_course(y) -> takes(x,y))" has a corresponding abstract tree:

```
All(['y'],
    Imp(Rel('maths_course',[Var('y')]),
        Rel('takes',[Var('x'),Var('y')])))
```

The formula "∀s (takes(s,c) <-> member(s,sl))" is first split into the conjunction of two formulas

"∀s ((takes(s,c) -> member(s,sl)) & (member(s,sl) -> takes(s,c)))"

which has the corresponding abstract tree

```
All(['s'],
    And(Imp(Rel('takes',[Var('s'),Var('c')]),
            Rel('member',[Var('s'),Var('sl')])),
        Imp(Rel('member',[Var('s'),Var('sl')]),
            Rel('takes',[Var('s'),Var('c')]))))
```

4. A program

The type definitions are extended by

```
TYPEDEF program = list(relation)

TYPEDEF relation = Reldef(name,backdef,forwarddef).

TYPEDEF backdef = list( assertion ! limplication ).

TYPEDEF forwarddef = list(rimplication).

TYPEDEF assertion = Assert(list(name),list(term)).

TYPEDEF limplication = Limp(list(name),list(term),formula)

TYPEDEF rimplication = Rimp(list(name),list(term),formula)
```

Each relation definition definition set is split into two parts, one for the forward implications (->) and one for the backward implications (<-). This corresponds to the types 'forwarddef' and 'backdef' above, each consisting of a list of assertions or implications.
The relation 'member'

```
        ¬ member(x,[]).
        member(x,[y|z]) <-> x=y or member(x,z).
```

is transformed to

```
[Reldef('member',[Limp(['x','y','z'],
                       [Var('x'),Dstruct('.',[Var('y'),Var('z')])],
                       Or(Eq(Var('x'),Var('y')),
                          Rel('member',[Var('x'),Var('z')]))),
                  [Rimp(['x'],
                       [Var('x'),Dstruct('[]',[])],
                       False),
                   Limp(['x','y','z'],
                       [Var('x'),Dstruct('.',[Var('y'),Var('z')])],
                       Or(Eq(Var('x'),Var('y')),
                          Rel('member',[Var('x'),Var('z')])))])]
```

5. The environment

The values of the variables during a computation are found in the environment.

```
TYPEDEF environment = list(<Context(loc,context),loc>)

TYPEDEF context = list(Binding(name,value))

TYPEDEF value = <term,loc> ! STAR ! UNBOUND
```

where loc is an integer.

A formula is always considered in a certain context, and this context is conveniently referred by a location (an integer).

If we have the formula

$\exists x,y,z\ (\ \exists z,x\ (\ q(x,y,z) \rightarrow r(x,y)\ )\ \&\ s(x,z)\ \&\ p(x,y,z))$

we will in a procedural interpretation get the following:
After having "performed" the outermost existential quantifier the variables x,y and z are known. We then have

[]	
x	UNBOUND
y	UNBOUND
z	UNBOUND

10 [<Context([], [Binding('x',UNBOUND), Binding('y',UNBOUND), Binding('z',UNBOUND)]),10>]

and the current context is 10.
If we then immediatly perform the inner quantifier we get

10	
z	UNBOUND
x	UNBOUND

11 [<Context(10, [Binding('x',UNBOUND), Binding('y',UNBOUND)]),11>,

[]	
x	UNBOUND
y	UNBOUND
z	UNBOUND

10 <Context([], [Binding('x',UNBOUND), Binding('y',UNBOUND), Binding('z',UNBOUND)]),10>]

The value of variable 'x' in context 11 we quite naturally find in context 11, but the variable 'y' can't be found in context 11. We then follow the **static chain** which is given by the first argument of a context. The static chain of context 11 is 10, while context 10 does not have any static chain. We have a special notation for the value of a variable in a certain

context; the variable 'x' in context l1 is written <Var('x'),l1>.

6. The 'gepr' machine

The basis for the 'gepr' machine is a state transition system. We have a set of state transition rules

$$state_i \dashrightarrow state_{i+1}$$

The behaviour of the machine is described by the transitive closure of the transition relation, which we write as '-->*' and is defined by

$$terminal\text{-}state \dashrightarrow^{*} terminal\text{-}state$$

$$state \dashrightarrow^{*} terminal\text{-}state \text{ if } state \dashrightarrow^{*} state_1 \text{ and } state_1 \dashrightarrow terminal\text{-}state$$

Each 'state' consists of four 'registers':

G Goal-stack
E Environment
P Program
R Resumption register to handle backtracking

We have already shown the types of E and P, and R will be elaborated when we come to 'or' in formulas.
The goal-stack is perhaps the most complex type of the 'gepr'-machine. It contains information in very goal directed manner of what that has to be done. Since the machine has two major modes of execution, backward proof and forward proof mode, the goal-stack contains two types of items. (The special "goal" Fail may also occur on the goal-stack).

```
TYPEDEF goalstack = list(goal)
TYPEDEF goal = B(<formula,loc>, conclusion_environment) !
               F(list(<formula,loc>), <formula,loc>, conclusion_environment) !
               Fail
TYPEDEF conclusion_environment = ...almost the same as environment...
```

The backward proof mode of execution corresponds roughly to the normal "Prolog" mode of execution, while the forward mode is needed to handle implications in formulas.
We start with explaining the backward proof mode, but first we show how to initialize the 'gepr' machine.

If we want to evaluate a formula 'f' in a program 'p', the 'gepr'-machine starts with

```
        G                E   P  R
([B(<f,[]>,[])], [], p, [])
```

i.e. the goal-stack G contains just the item B(<f,[]>,[]). This means that we are going to perform a **backward proof** of f in an empty context. The context is empty because we assume that all variables are explicitly quantified in the formula, and contexts will be created for those variables.
The second argument of B (the conclusion_environment) is also empty initially.
The environment is empty since we don't have any bindings of any variables when we start.
The resumption register is empty since we have no backtracking points.

A successful final state of the 'gepr' machine is

```
([], e, p, r)
```

where the bindings of the variables are found in the environment e. If we, for some reason, want another solution, the machine may be restarted again with the information in the resumption register. This is shown below in the 'Fail'-transition.
It may also happen that the whole computation fails. This is the case if the final state is

```
([Fail|gs], e, p, [])
```

We are now ready to show the state transition rules, i.e. rules that convert one state of the 'gepr' machine to the next. Each rule corresponds to a rule in a natural deduction system.

7. Backward proof

7.1. And

If we in a natural deduction system are going to prove "f1 & f2", we first prove f1 separately and then prove f2. Due to the '&' introduction rule we have

```
   f1   f2
  ---------
   f1 & f2
```

which is read backwards in a backward proof. In the 'gepr' machine this corresponds to the state transition

```
([B(<And(f1,f2),l>,ce)|gs],      e, p, r)  -->
([B(<f1,l>,ce),B(<f2,l>,ce)|gs], e, p, r)
```

Since no new variables are introduced, the context 'l' is unchanged. The rest of the goal-stack is 'gs', and is left untouched by the transition.

7.2. Or

If we want to prove "f1 or f2" we may either prove f1 or f2:

```
      f1                      f1
  ----------     or       ----------
   f1 or f2                f1 or f2
```

And the corresponding transition

```
([B(<Or(f1,f2),l>,ce)|gs],e,p,r) -->
([B(<f1,l>,ce)|gs]       ,e,p,[GEP([B(<f2,l>,ce)|gs],e,p)|r])
```

The computation continues with a backward proof of f1, while we have saved the contents of the registers G, E and P on the resumption stack for the alternative computation. If we, for any reason, fail with proving f1, we may retry and prove f2. A failure of a computation is indicated by a special "goal" called 'Fail' on the goal-stack. Although we have not yet shown how a 'Fail' gets to the goal-stack, we here show how the 'gepr' machine reacts.

```
([Fail|gs],e, p,[GEP(gs1,e1,p1)|r1]) -->
(gs1,      e1,p1,r1).
```

It may however occur that the resumption stack is empty. We then don't have any valid alternatives and the whole computation has failed.

```
([Fail|gs],e,p,[])      no solution!
```

The type of the resumption stack is called 'dump':

```
 TYPEDEF dump = list(GEP(list(goal),environment,program))
```

7.3. False

The simplest way to fail in a backward proof is to find an explicit 'False' in the goal-stack. The 'gepr' machine simply converts this to Fail.

```
([B(<False,_>,_)|gs],e,p,r) -->
([Fail],             e,p,r).
```

7.4. Exists

If a group of variables are existentially quantified, we allocate a storage for those variables and initialize them to unbound. This will effect the environment. In natural deduction this becomes

```
   f(v)
--------
∃v(f(v))
```

and the proof may continue backwards from 'f(v)'.

```
([B(<Exist(vs,f),l>,ce)|gs],e,p,r) -->
([B(<f,l1>,ce)|gs],         e1,p,r)   if  newenv(vs,l,e,UNBOUND)=[l1,e1].
```

where e1 is the new environment and l1 is the location of the new context. The type of the function newenv is

```
    TYPEOF newenv(list(name),loc,environment,value)=[loc,environment]
```

For example, the first environment shown on page 5 could have been created by newenv(['x','y','z'],[],[],UNBOUND).

7.5. For all

A group of variables may alternatively be universally quantified. In the natural deduction system we mark the variables (with a star) so they cannot become bound.

```
   f(*v)
----------
∀v(f(v))
```

The 'gepr' machine makes almost the same things as for an existential quantification, but all the variables are bound to the special value 'STAR'.

```
([B(<All(vs,f),l>,ce)|gs],e,p,r) -->
([B(<f,l1>,ce)|gs],       e1,p,r)   if  newenv(vs,l,e,STAR)=[l1,e1].
```

When such a variable is found during a unification, it cannot be bound, and

will remain having the value 'STAR'. This is quite natural since a universally quantified variable cannot be restricted to a special value.

7.6. Equality

The special atomic relation '=' also gets a special treatment. In a backward proof however, the treatment seems to be quite normal. We just invoke unification:

```
We have two cases
    ([B(<Eq(t1,t2),l>,ce)|gs],e,p,r) -->
    gepr(gs,e1,p,r)
        if  unify([<t1,l>],[<t2,l>],ce,e) = e1 and
            e1 /= Fail
or
    ([B(<Eq(t1,t2),l>,ce)|gs],e,p,r) -->
    gepr([Fail],e,p,r)
        if  unify([<t1,l>],[<t2,l>],ce,e) = e1 and
            e1 = Fail
```

The function 'unify' returns the new environment in case of success, otherwise it returns the constant 'Fail'. The type of 'unify' is

```
TYPEOF unify(list(<term,loc>),list(<term,loc>),
            conclusion_environment, environment) = (environment ! Fail)
```

The unification differs from a standard unification in several ways.
When we want to get the value of a variable, we always first look in the ce ('conclusion_envirionment') for reasons which will be explained later. If the variable is unbound there we then use the normal environment.
As already mentioned, the STAR variables are not allowed to be bound to be bound to anything, and no variable is allowed to become bound to a STAR variable. There is however one exception when we can convert a proof made with a binding to a STAR variable to a proof without such a binding. In natural deduction we may have

```
      Π
      :
    B(x,*y)
  -----------
  ∃x B(x,*y)
```

where B(x,*y) is an arbitrary complex formula containing x and y. If we have to assume x=*y in order to perform the proof Π, we can convert the whole proof Π to a new proof that does not need that assumption. Since

```
      Π2
      :
    B(*y,*y)
  -----------
  ∃x B(x,*y)
```

is a valid step in natural deduction, and the proof Π2 does not contain any assumptions on *y, the formula "∃x B(x,*y)" is valid. What is crucial is that the existentially quantified variable must 'declared' after the universally quantified variable (the 'STAR' variable).
This mechanism allows us to conclude that

```
    ∀x∃y (x=y) is true
```

while

```
    ∃x∀y (x=y) is false
```

Which one of the variables that is 'declared' first is easily tested by comparing the location numbers of the variables. In this case it is important to follow the static chain to get the true location of the variable.

Although not strictly necessary we have also chosen to implement unification so that it can handle cyclic structures [Ha81].

7.7. Atomic relation

In a backward proof when an atomic relation is encounted, we need the program to find the definition of that relation. There may be none, one or several such relation definitions. We take them in the defined order of the backward definition set and perform a unification, which may change the environment. In natural deduction we write

$$\frac{\forall x1,..xn(\ relation\_name(q1,..,qn) \leftarrow f\ ) \qquad \begin{matrix}\Pi \\ : \\ f\end{matrix} \qquad q1=r1\ ..\ qn=rn}{relation\_name(r1,..,rn)}$$

Above the line we have three parts: a part of the relation definition set, the formula f (which is the right hand side of the relation definition), and a group of equalities generated during unification. If are able to prove f in backward mode we may then conclude the formula that is written under the line.
In the 'gepr' machine we first find all the definitions of the relation. This is done by the function 'getbackdef'. For each of the relation definitions found we stack a unification request on the resumption stack. This is done by the function 'newdumpb'. Finally we invoke failure so that the top item (if any) of the resumption stack will be used.

```
([B(<Rel(rn,ts),l>,ce)|gs],e,p,r) -->
([Fail],e,p,r1)              if getbackdef(p,rn) = stmts and
                                newdumpb(ts,stmts,l,ce,gs,e,p,r)=r1
```

Although it looks a bit complicated, the function 'newdumpb' is very simple:

```
TYPEOF newdumpb(list(term),backdef,loc,conclusion_environment,list(goal),
                environment,program,dump)=dump
newdumpb(_,[],_,_,_,_,_,r)=r.
newdumpb(ts,[stm|stms],l,ce,gs,e,p,r) =
    [GEP([B(<Unify(ts,stm),l>,ce)|gs],e,p)|newdumpb(ts,stms,l,ce,gs,e,p,r)].
```

The type of the variable 'stmts' above is 'backdef' (see page 4), which means that we actually have two types of clauses: assertions and relations. We don't show the assertions here since they are identical to a relation with an always true right hand side.
When the 'gepr' machine finds a unification request on top of the goal-stack it first allocates space for the new variables and then performs the unification.

```
We then have two cases:
    ([B(<Unify(ts1,Limp(vs,ts2,f)),l1>,ce)|gs],e,p,r) -->
    ([B(<f,l2>,ce)|gs],                              e2,p,r) )
          if newenv(vs,[],e,UNBOUND) = [l2,e1] and
             unify(ctermlist(ts1,l1),ctermlist(ts2,l2),ce,e1) = e2 and
             e2/=Fail
or
    ([B(<Unify(ts1,Limp(vs,ts2,f)),l1>,ce)|gs],e,p,r) -->
    ([Fail],                                         e2,p,r) )
          if newenv(vs,[],e,UNBOUND) = [l2,e1] and
             unify(ctermlist(ts1,l1),ctermlist(ts2,l2),ce,e1) = e2 and
             e2=Fail

The type of 'ctermlist' is
 TYPEOF ctermlist(list(term),loc)=list(value)

ctermlist([],_) = [].
ctermlist([a|as],l) = [<a,l>|ctermlist(as,l)].
```

7.8. Implies

Finally, here is the rule that changes the execution mode from backward proof to forward proof. In natural deduction we have

```
          f1[1]
          :
          Π
          :
          f2
       --------[1]
        f1->f2
```

The expression under the line is true if we by starting by assuming f1 can prove f2. This proof is marked with Π in the figure above. During that compution certain conclusions may have been drawn, which obviously depend on the assumption f1, and they must therefore be discharged after the subproof. This is schematically indicated by "[1]" in the figure. We solve this problem by having a local 'conclusion environment' for all subcomputations. By this, local conclusions don't effect the global status of the computation (the environment).
In the 'gepr' machine we have

```
    ([B(<Imp(f1,f2),l>,ce)|gs],e,p,r) -->
    ([F([<f1,l>],<f2,l>,ce)|gs],e,p,r)
```

The formula f1 is called the "premise goal". In general we may have a list of "premise goals", but at start there is just one.
We are now ready for

8. Forward proof

The action in forward proof mode generally depends on the form of the premise goal.

8.1. And

The first rule is a simple rewrite

```
([F([<And(f1,f2),l0>|fr],<cf,l>,ce)|gs],e,p,r) -->
([F([<f1,l0>,<f2,l0>|fr],<cf,l>,ce)|gs],e,p,r)
```

which extends the list of premise goals.

8.2. Or

If the first premise goal is an 'or'-form we have

```
                                                  f1[1]    f2[1]
                                                  :        :
f1 or f2                                          Π1       Π2
   :                                              :        :
   Π                                 f1 or f2     cf       cf
   :        which we convert to     ------------------------------[1][2]
  cf                                              cf
```

That is, to prove that f1 or f2 implies cf we have to prove that f1 implies cf **and** f2 implies cf. The conclusions drawn at these subcomputations must as usual be removed after the computation.
This means that

```
([F([<Or(f1,f2),l0>|fr],<cf,l>,ce)|gs],e,p,r) -->
([F([<f1,l0>|fr],<cf,l>,ce),F([<f2,l0>|fr],<cf,l>,ce)|gs],e,p,r)
```

8.3. Exist

If variables are existentially quantified in the premise goal we get

```
                f(*v)[1]
                  :
                  Π
                  :
   ∃v(f(v))      cf
   ---------------------[1]
          cf
```

We try to perform a proof of 'cf' starting from the premise goal f(*v).
All the existentially quantified variables have the value STAR, and have to follow the rules of a STAR variable. In 'gepr' we get

```
([F([<Exist(vs,f),l0>|fr],<cf,l>,ce)|gs],e,p,r) -->
([F([<f,l1>|fr],<cf,l>,ce)|gs],e1,p,r) if newenv(vs,l0,e,STAR) = [l1,e1]
```

8.4. For all

Similarly for universal quantification we have

```
     ∀v(f(v))
    ----------
       f(v)
        :
        Π
        :
       cf
```

and for 'gepr'

```
([F([<All(vs,f),l0>|fr],<cf,l>,ce)|gs],e,p,r) -->
([F([<f,l1>|fr],<cf,l>,ce)|gs],e1,p,r) if newenv(vs,l0,e,UNBOUND) = [l1,e1]
```

8.5. False

If the premise is false we can end the subcomputation with success without further computations.

```
 False
-------
  cf
```

And for 'gepr' the computation continues with 'gs':

```
([F([<False,l0>|fr],<_,_>,_)|gs],e,p,r) -->
(gs,                              e,p,r)
```

8.6. Implies

Even in forward proof mode an implication can be found in the premise goal.

```
f1->f2                                  Π1
  :                                     :
  Π       which is converted to         f1        f1-> f2
  :                                     ----------------------------
 cf                                             f2
                                                :
                                                Π2
                                                :
                                                cf
```

We first try to perform the backward proof Π1, and then the forward proof Π2.

```
([F([<Imp(f1,f2),l0>|fr],<cf,l>,ce)|gs],e,p,r) -->
([B(<f1,l0>,ce),F([<f2,l0>|fr],<cf,l>,ce)|gs],e,p,r)
```

8.7. Equality

If an equality is found in the premise goal-list, we may use this equality anywhere in the subproof. The situation is very different from ordinary unification, almost the opposite. We first look for the value of a variable in the normal environment (e), and only if the value is UNBOUND or STAR we look in the conclusion environment (ce). If it necessary to bind a variable in order to succeed in the conclusion unification, the new value is only stored in the conclusion environment.
If the conclusion unification fails, it must be remembered that a false premise implies everything, so we have actually succeeded!

```
Two cases:
     ([F([<Eq(t1,t2),l0>|fr],<cf,l>,ce)|gs],e,p,r) -->
     ([F(fr,<cf,l>,ce1)|gs],e,p,r)
        if  cf /= <False,_>
        and concunify([<t1,l0>],[<t2,l0>],ce,e) = ce1
        and ce1/=Fail
or
     ([F([<Eq(t1,t2),l0>|fr],<cf,l>,ce)|gs],e,p,r) -->
     (gs,                                   e,p,r)
        if concunify([<t1,l0>],[<t2,l0>],ce,e) = ce1
        and ce1=Fail
where
TYPEOF concunify(list(<term,loc>),list(<term,loc>), conclusion_environment,
                 environment) = (conclusion_environment ! Fail)
```

In the code above we check so that the conclusion formula isn't an explicitly 'False' formula. If concunify succeeds we would otherwise be sure that the computation whould fail. To avoid this we test for that special case, and we treat it separately. In this case the only solution for success is to use some sort of 'negative' unification which generates assumptions like 'x≠17'. The advantage of this is a wider domain of executable programs, while the disadvantage is increased nondeterminism. In this paper we will not elaborate this mechanism further.

8.8. Atomic relation

When we find an atomic relation in the premise goal-list, we try to find its definition set in the program. If there are several definitions we take the first and save the rest on the resumption stack. The case in natural deduction when we have found one definition:

$$\frac{\forall x(\ \text{relation\_name}(q1,..,qn) \leftarrow f) \qquad \text{relation\_name}(r1,..,rn) \qquad q1{=}r1,..,qn{=}rn}{\begin{array}{l} f \\ : \\ \Pi \\ : \\ cf \end{array}}$$

Above the line we have three parts: a part of the relation definition set, the premise goal, and a group of equalities generated during unification. We then have to perform the proof Π.
As in backward proof mode we first find the definition set and stack the alternative definitions on the resumption stack. We then invoke failure so the top item on the resumption stack will be used.

```
     ([F([<Rel(rn,ts),l0>|fr],<cf,l>,ce)|gs],e,p,r) -->
     gepr([Fail],e,p,r1)
        if  getforwarddef(p,rn) = stmts and
            newdumpf(ts,stmts,l0,fr,cf,l,ce,gs,e,p,r)=r1
where
 TYPEOF newdumpf(list(term),forwarddef,loc,list(<formula,loc>),<formula,loc>,
                 loc,conclusion_environment,list(goal),environment,
                 program,dump)=dump.
 newdumpf(_,[],_,_,_,_,_,_,_,_,r)=r.
 newdumpf(ts,[stm|stms],l0,fr,cf,l,ce,gs,e,p,r) =
 [GEP([F([<Unify(ts,stm),l0>|fr],<cf,l>,ce)|gs],e,p)|
          newdumpf(ts,stms,l0,fr,cf,l,ce,gs,e,p,r)].
```

In 'gepr' machine we may encounter 'unifications' in forward proof mode. The unification may either fail or succeed.

We have two cases:

```
     ([F([<Unify(ts1,Rimp(vs,ts2,f)),l0>|fr],<cf,l1>,ce)|gs],e,p,r) -->
     ([F(fr,<cf,l1>,ce)|gs],_,_,r)
        if  newenv(vs,[],e) = [l2,e1]
        and unify(ctermlist(ts1,l0),ctermlist(ts2,l2),ce,e1) = e2
        and e2=Fail
or
     ([F([<Unify(ts1,Rimp(vs,ts2,f)),l0>|fr],<cf,l1>,ce)|gs],e,p,r) -->
     ([F([<f,l2>|fr],<cf,l1>,ce)|gs],e2,p,r)
        if  newenv(vs,[],e) = [l2,e1]
        and unify(ctermlist(ts1,l0),ctermlist(ts2,l2),ce,e1) = e2
        and e2/=Fail
```

9. Discussion

It is important to stress that we have not devised a complete theorem prover. This means that there is a domain of formulas that we are unable to prove. We claim however that these formulas usually are not computationally useful. To include them in the set of formulas we can prove would increase the nondeterminism and degrade the system performance.
One improvement towards a more complete system was the 'negative' unification which was discussed on page 14. Since we have not yet been able to use the system for large problems, we not yet sure whether this complication would be worthwile. We do however already know that there is a wide domain of programs where the system has proven to be quite useful.

10. References

[La63] Landin, P J, The Mechanical Evaluation of Expressions, Computer Journal, 6 (4), 308-20, 1963

[He80] Henderson, P, Functional Programming, Prentice-Hall International, 1980

[Ha81] Haridi, S, Logic Programming Based on a Natural Deduction System, thesis, Royal Institute of Technology 1981, Stockholm

[HHT82] Å Hansson, S Haridi, S-Å Tärnlund, Properties of a Logic Programming Language, in Logic Programming edited by K L Clark and S-Å Tärnlund, Academic Press 82

CONTEXTUAL GRAMMARS IN PROLOG

Paul SABATIER

Laboratoire d'Automatique et de Linguistique
Universite Paris 7
Tour Centrale 9 eme
2 Place Jussieu
75005 PARIS

ABSTRACT

We present a formalism and a technique by which left and/or right contextual constraints can be easily expressed and computed efficiently in Prolog grammars (avoiding transport of variables): the Contextual Grammars (CG), interpreted in PROLOG II.

Each rule has the form:

NT -> CONTEXT BODY.

where NT is a non-terminal symbol. BODY is a sequence of one or more items separated by blanks. Each item is either a non-terminal symbol, a terminal symbol or a condition. Symbols and conditions are terms (as in Metamorphosis or Definite Clauses grammars); BODY may be empty.

If CONTEXT is not empty, it has the form:

{ L # R }

L and R are sequences of non-terminal and/or terminal symbols separated by points. We read it as:

Apply NT if, in the derivation tree,
1) L precedes NT, and
2) R follows NT.

L or R may be empty.

For example, the following is a sample contextual grammar (terminal symbols are in brackets, and conditions are preceded by "+"):

```
sentence(S)   -> np(_)  vp(S).
np(X.Y)       -> noun(X)  [and]  noun(Y).
np(X)         -> noun(X).
noun(day)     -> [day].
noun(night)   -> [night]
vp(S)         -> verb(S).
vp(S)         -> verb(S)  preposition  np(_).
preposition   -> [with].
```

```
(a)  verb(alternate(X,Y)) -->  { np(X.Y) # }
                               +different(X,Y)
                               [alternate].

(b)  verb(alternate(X,Y)) -->  { noun(X) # [with].noun(Y) }
                               +different(X,Y)
                               [alternates].
```

The sentences produced/analysed from (a) are:

day and night alternate.
night and day alternate.

and from (b):

day alternates with night.
night alternates with day.

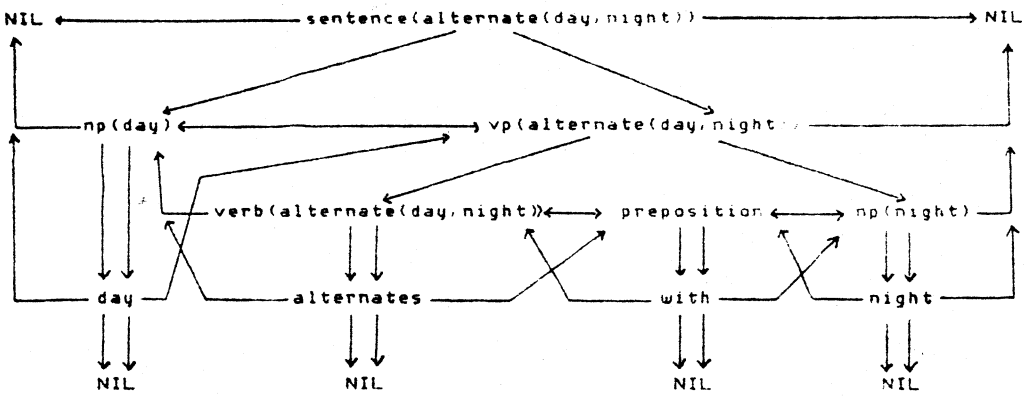
The technique consists in building, during the parsing, an internal derivation graph G containing the sufficient information to recover the context whenever a contextual constraint must be satisfied before the rule must be applied. To each node Ni (corresponding to a non-terminal or terminal symbol) of G, are associated four nodes Nj, Nk, Nl and Nm:

- Nj is the left sibling of Ni; Nj is the left sibling of the parent of Ni if Ni has no left sibling;
- Nk is the first child of Ni; Nk is NIL if Ni has no children;
- Nl is the last child of Ni; Nl is Nk if Ni has one child; Nl is Nk is NIL if Ni has no children;
- Nm is the right sibling of Ni; Nm is the right sibling of the parent of Ni if Ni has no right sibling;

The right sibling and the left sibling of the axiom-symbol of the grammar are NIL.

Here, for example, is the final derivation graph of the sentence:

day alternates with night.



Contextual constraints are computed directly from G. When any part of a context is not yet known (as for example right context of a symbol in a left-to-right parser), the computation is delayed by means of the GELER (FREEZE) predicate.

CURRENT TRENDS IN LOGIC GRAMMARS

Veronica Dahl
Computing Sciences Department
Simon Fraser University
Burnaby, BC V5A 1S6

Abstract

This paper surveys several logic grammar formalisms, relates them to some recent trends in linguistics and advocates the use of logic grammars for natural language processing. Contrary to many recent approaches that resort to augmenting essentially context-free grammars, it also tries to make a case for not outruling context-sensitivity or transformations. Finally, it presents a new logic grammar formalism jointly developed by Michael McCord and the author, the main features of which are: a metagrammatical treatment of coordination that relieves the grammar writer from having to describe coordinating rules explicitly; a modular treatment of semantics based upon simple information given locally to each rule, and an automated building-up of the sentence's representation structure.

1. Introduction

Among the computational formalisms for describing and processing language, logic grammars have been drawing attention since their introduction in 1975 (Colmerauer 1975).

Logic grammars resemble type-0 grammars, except that the grammar symbols may have arguments, and that procedures may be invoked from the rules (e.g. to serve as applicability constraints). Derivations involve unifying (Robinson 1965) symbol strings rather than just replacing them. Since the logic grammar formalism is a part of the Prolog programming language (Colmerauer 1975, Pereira L et al, 1978), logic grammars written to describe a language can be interpreted by Prolog as analysers for that language. Thus relieved from the operational concerns of parsing, the user can develop very clear and concise "analysers" just by writing a set of logic grammar rules that describe a language and giving it to Prolog. Logic grammars have been favourably compared with a widely used formalism for processing language: augmented transition networks (ATNs) introduced in 1970 (Woods 1970). They have been argued to be clearer, more concise and in practice more powerful, while at least as efficient, as ATNs (Pereira & Warren, 1980).

The first sizable application for logic grammars was a Spanish/French consultable database system (Dahl 1977,1981,1982) which was later adapted to Portuguese by H. Coelho and L. Pereira

(1), and to English, by F. Pereira and David Warren (2); and has since inspired the development of several other applications (e.g. Coelho 1979, McCord 1980, F. Pereira & Warren 1981). This system has been shown to be comparable in efficiency with the LUNAR system (Woods et al., 1972), (cf. Pereira & Warren, 1980, p.276), thus joining the appeal of practical feasibility to the elegance and expressive power of the logic grammar approach. Further logic grammar applications include (Silva et al. 1979, Simmons and Chester 1979, Sabatier 1980, Pereira et al. 1982). However the experience gained in the aforementioned applications has motivated the development of alternative logic grammar formalisms, some restricting and others augmenting the power of the original formalism as described in (Colmerauer 1975).

This paper attempts to fill a gap by examining the evolution of logic grammars, comparing the alternative proposals, and discussing them with respect to recent trends in both theoretical linguistics and natural language processing. It also motivates and briefly presents a new logic grammar formalism, called "modifier structure grammars" (MSGs), developed jointly by Michael McCord and the author (Dahl and McCord 1983). Its main features are: a metagrammatical (user-invisible) treatment of coordination, a modular treatment of semantics, and an automatic build-up of the parsed sentence's representation.

---

(1) Personal Communication, 1978.
(2) Personal Communication, 1980.

Section 2 describes logic grammars in intuitive, user-biased terms. Section 3 presents different types of logic grammars; Section 4 compares them with respect to expressive power, in particular through the example of how they allow to express movement of constituents. Section 5 discusses pros and cons of choosing relatively evolved grammar formalisms, and makes a case for choosing logic grammars independently of the degree of evolution needed. Section 6 briefly presents our new logic grammar formalism (MSGs), and Section 7 contains some concluding thoughts.

2. What is a logic grammar?

Logic grammars can be thought of as ordinary grammars, in which the symbols may have arguments. These arguments are either constants, variables or functional expressions, and the fact that they may include variables implies that substitutions are sometimes needed in order to apply a grammar rule. For instance, consider the following grammar:

```
1)  Sentence (fact(F)) --> proper-noun(N), verb(N,F).

2)  proper-noun (mary) --> [mary].
3)  proper-noun (john) --> [john].

4)  verb (N, laughs(N)) --> [laughs].
5)  verb (N, smiles(N)) --> [smiles].
```

in which (as throughout this paper) constants are in lower-case

letters, variables start with a capital, consecutive terminal symbols are represented as square-bracketed lists, and non-terminals are in lower-case letters. The comma stands for "concatenation", and the end of a rule is signalled by a period. Having defined these rules to Prolog, if we now, for instance, want to analyse the sentence "Mary smiles", we merely write the question:

```
? sentence(X,"mary laughs", [ ])
```

This amounts to a request that Prolog find a value for X that represents the surface form "Mary smiles" with respect to this grammar. What happens in the Prolog execution of the parsing procedure can be summarized in the top-down, left-to-right derivation tree depicted in Fig. 1, where each rule application is labelled by the identification of the rule involved and by the set of substitutions of terms for variables that are needed in order to apply the rule.

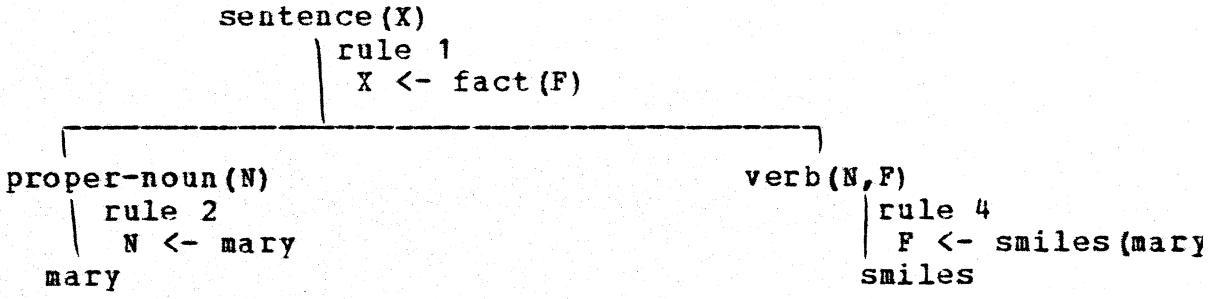


Figure 1. Derivation tree for "Mary laughs".

Only successful rule applications are shown here. Prolog backtracks upon unsuccessful ones. Through the substitutions employed, Prolog finds the representation

X = fact(smiles(mary))

for the sentence given.

Arguments allow for information to be shared by various grammar symbols, and to be carried along a derivation. In our example, they serve to build up a desired representation for a surface sentence. In procedural terms, grammar symbols can be thought of as producers and consumers of structure: the "proper-noun" symbol produces the value "mary", which is then consumed by "verb" in order to produce the structure "smiles(mary)", from which the final structure "fact (smiles (mary))" is constructed by "sentence".

Other uses in natural language processing include: syntactic and semantic checks (e.g. gender and number, semantic type or class), carrying extraposed constituents across phrases, etc.

For instance, we may check semantic accord by declaring Mary and John to be of type "human", and requiring that the arguments of "laughs" and "smiles" also be human. We use a functional symbol "-", in infix notation (allowed by Prolog) in order to introduce this semantic information. The above grammar becomes:

```
1)  sentence (fact(F)) --> proper-noun(N), verb(N,F)

2)  proper-noun (human-mary) --> [mary].
3)  proper-noun (human-john) --> [john].

4)  verb (human-N, laughs(N)) --> [laughs].
```

5) verb (human-N, smiles(N)) --> [smiles].

(Notice that variable names are local to each rule - i.e., variables with the same name are unrelated if they belong to different rules.)

We have enforced semantic agreement through unification, i.e., in the Prolog matching of terms. Proper nouns introducing non-humans now fail to be coupled with such verbs as "laughs" and "smiles".

Another way is through procedure calls, allowed in logic grammars in the form of Prolog calls (that we note between brackets). For instance, rule 4) could have instead been replaced by:

4) verb (N, laughs(N)) --> [laughs], human(N) .

and we would have added a Prolog definition for the procedure called, e.g.:

```
human(mary).
human(john).
```

More general procedures can, of course, be written in Prolog, e.g. "every child is human", noted:

```
human(x) :- child(x)
```

and read: "if x is a child then x is human".

3. Different types of logic grammars.

The first formulation of the parsing problem in terms of logic was obtained by A. Colmerauer and R. Kowalski, while trying to express Colmerauer's Q-System (Colmerauer 1973) in logic. This idea evolved into a very elegant and efficient Prolog implementation of metamorphosis grammars (Colmerauer 1975), that we shall call MGs.

An MG rule has the form:

$$S\ \alpha \text{ --> } \beta$$

where S is a nonterminal (logic) grammar symbol, α is a string of terminals and nonterminals, and β is like α except that it may also include Prolog procedure calls.*

Examples of such rules are:

```
a, [b] --> [b], a

verbroot (X), pluralmark --> [W], concat ([X],[s],W)
```

where a Prolog predicate concat (x,y,z) is assumed, that holds if z is the concatenation of x and y.

A special case of MGs was later included in DEC-10 Prolog

---

* The actual implementation in fact requires α to be a string of nonterminals, but we shall disregard the restriction since it has been shown (Colmerauer 1975) to involve no loss with respect to the full MG form.

(Pereira, Pereira & Warren 1978) and baptised definite clause grammars (DCGs). DCG rules have the form:

$$S \dashrightarrow \beta$$

where S and β are as above. All the rules presented in Section 2 are of this type.

The main motivation for introducing DCGs was ease of implementation coupled with no substantial loss in power (in the sense that DCGs can also basically describe type-0 languages - although less straightforwardly).

Extraposition grammars (XGs) (Pereira, to appear) were designed in order to refer to unspecified strings of symbols in a rule, thus making it easier to describe left extraposition of constituents.

XGs allow rules of the form

$$s_1 \ldots s_2 \text{ etc. } s_{k-1} \ldots s_k \dashrightarrow r$$

where the "..." specify gaps (i.e., arbitrary strings of grammar symbols), and r and the s_i are strings of terminals and non-terminals.

The general meaning of such a rule is that any sequence of symbols of the form

$$s_1 x_1 s_2 x_2 \text{ etc. } s_{k-1} x_{k-1} s_k$$

with arbitrary x_i s, can be rewritten into $r\ x_1\ x_2 \ldots x_{k-1}$

(i.e., the s_i's are rewritten into r, and the intermediate gaps (x_i's) are rewritten sequentially to the right of r. For instance, the XG rule:

relative-marker ... complement --> [that].

allows to skip any intermediate substring appearing after a relative marker * in the search for an expected complement, and then to subsume both marker and complement into the relative pronoun "that", which is placed to the left of the skipped substring.

The next section shows this rule at work in a parsing context.

Restriction grammars (RGs) (Hirschman & Puder, 1982) are not, strictly speaking, logic grammars, since the grammar symbols may not include any arguments. But they are implemented in Prolog and provide an instance of what seems to be a popular tendency in natural language processing nowadays: they involve sets of context-free definitions augmented with grammatical constraints or restrictions. In RGs, these appear in the form of procedures interleaved among the context-free definitions.

For instance, the RG rule

---

* i.e., a symbol that announces the beginning of a relative clause.

predicate ::= verb, object, verb-object

states that "predicate" can be rewritten into "verb object", provided that the "verb-object" restriction is satisfied (this restriction could for instance state that if the object is nil, the verb must be intransitive). Restrictions need to be defined separately, using such available primitives to traverse the tree as "up", and "down"; and explicitly stating parameters for its starting point in the tree and in the word stream.

4. Expressive power of each grammar formalism.

What are the consequences of choosing one of these grammar formalisms to write a natural language processor? From a theoretical point of view, the power of MGs, DCGs and XGs is similar in that they can all serve to describe type-0 languages. From a practical point of view, however, their possibilities differ. In this section we shall illustrate this point by studying how easily and concisely each of these formalisms allows to describe those rules involving constituent movement and ellision. RGs, although not dealing specifically with movement, are also considered.

4.1 Movement rules and MGs.

Let us consider for instance the noun phrase:

the man that John saw

which can be thought of as the surface expression of the more

canonical form:

the man [John saw the man],

where the second occurrence of "the man" has been shifted to the left and subsumed into the relative pronoun "that".

A simple grammar for (very restricted) sentences in canonical form could be:

(1) sentence --> noun-phrase, verb-phrase.

(2) noun-phrase --> determiner,noun,relative.

(3) noun-phrase --> proper-name.

(4) verb-phrase --> verb.

(5) verb-phrase --> trans-verb,direct-object.

(6) relative --> [ ].

(7) direct-object --> noun-phrase.

(8) determiner --> [the].

(9) noun --> [man].

(10) proper-name --> [john].

(11) verb --> [laughed].

(12) trans-verb --> [saw].

We shall successively modify this grammar (referred to as G in all that follows) in order to describe the relativization process within various logic grammar formalisms.

Within MGs, all we need is to add the following rules:

(6') relative --> relative-marker, sentence.

(5') verb-phrase --> moved-dobj, transitive-verb.

(13) relative-marker, noun-phrase,moved-dobj --> rel-pronoun,noun-phra

(14) relative-pronoun --> [that].

Figure 2 depicts the derivation tree for our sample noun phrase "the man that John saw". We abbreviate some of the grammar symbols. Rule numbers appear as left-hand side labels.

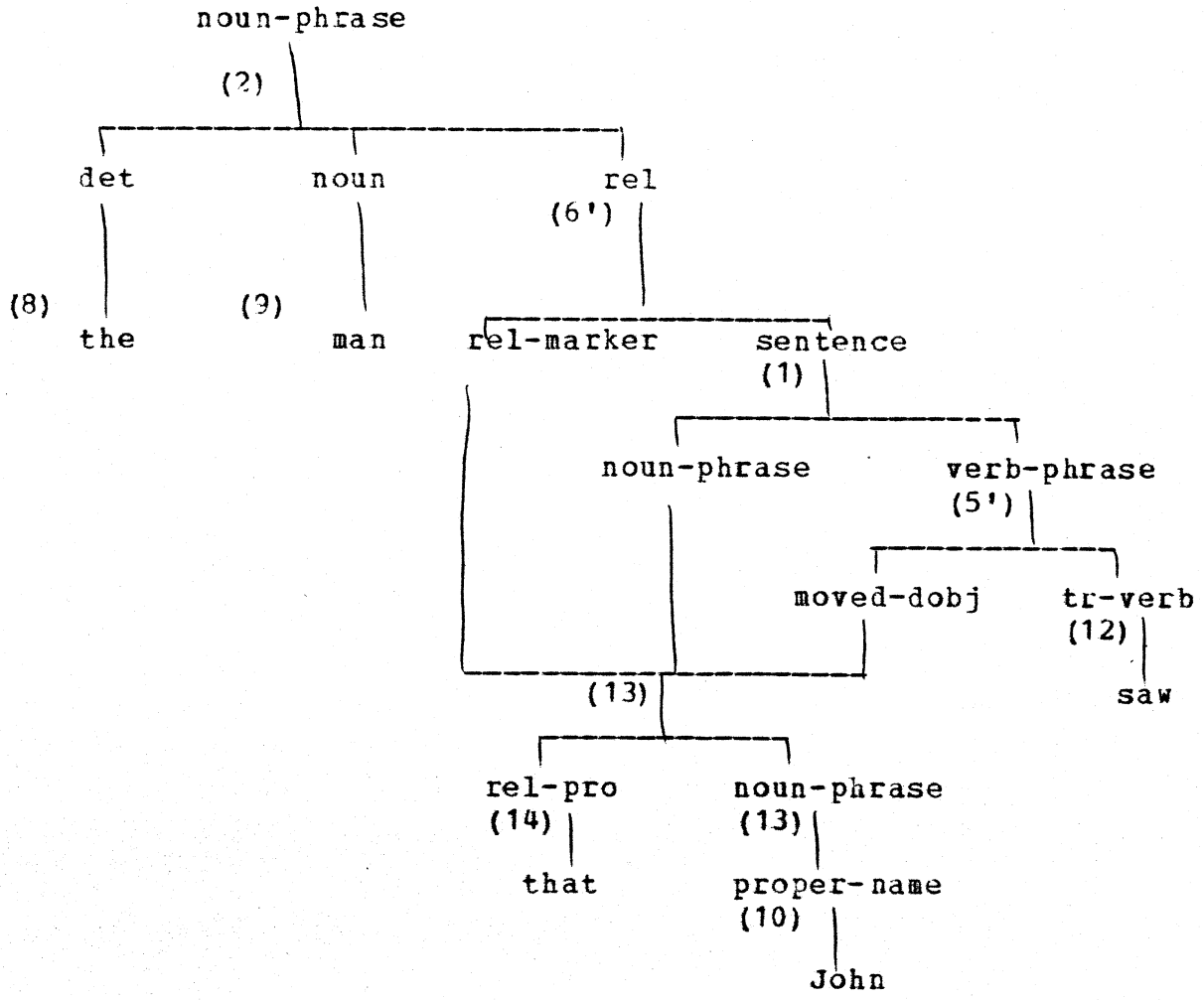


Figure 2. MG derivation tree for "The man that John saw".

Of course, for such a parse to be of any use, we need to construct a representation for the sentence while we parse it. But for the time being we shall ignore symbol arguments in order to concentrate upon the particular problem of moving constituents.

4.2. Movement rules and DCGs.

In terms of DCG rules, the simplest possible modification to the original grammar G is to allow a direct object to be ellided, e.g. by adding the rule:

(7') direct-object --> [ ].

But, because this rule lacks the contextual information found in (13), a direct object is now susceptible of being ellided even outside a relative clause. In order to prevent it, a usual technique is to control rule application by adding extra arguments. In our example, we only need to add a single argument that we carry within the sentence, verb-phrase and direct-object symbols, and that takes the value "nil" if the direct object in the verb phrase of the sentence is not ellided, and the value "ellided" if it is. The modified rules are the following:

(0) sentence --> sent(nil).

(1) sent(E) --> noun-phrase, verb-phrase(E).

(4) verb-phrase(nil) --> verb.

(5) verb-phrase(E) --> transitive-verb, direct-object(E).

(6') relative --> relative-pronoun, sent(E).

(7) direct-object(nil) --> noun-phrase.

(7') direct-object(ellided) --> [ ].

(13) relative-pronoun --> [that].

Figure 3 shows the DCG derivation tree for "The man that John saw laughed". Substitutions of terms for variables are shown as right-hand side labels.

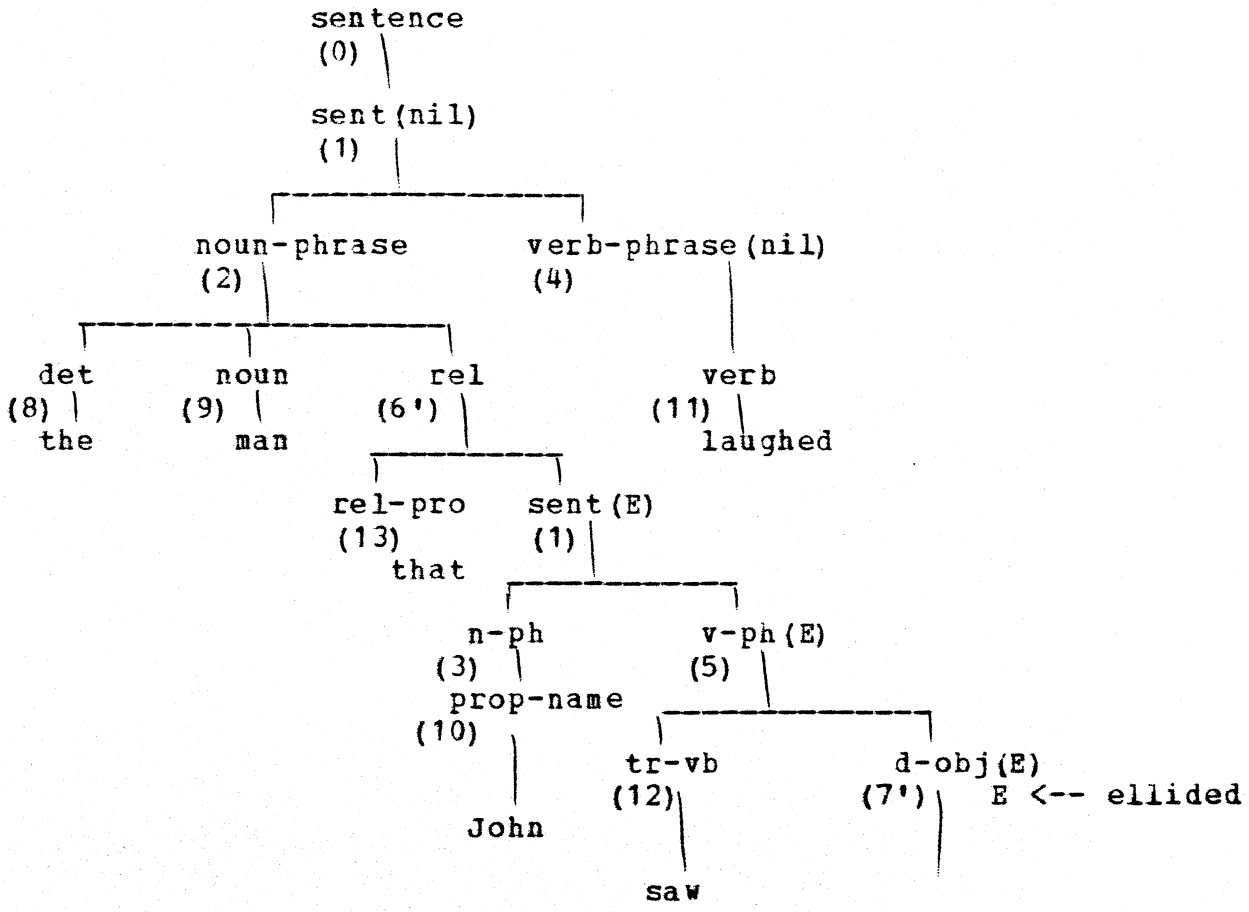


Figure 3. DCG derivation tree for the sentence "The man that John saw laughed".

4.3. Movement rules and XGs.

While, as we have seen, MGs express movement by actually moving constituents around, DCGs must carry all information relative to movements within extra arguments. XGs, on the other hand, can capture left extraposition in an economical fashion: by actually skipping intermediate substrings rather than shifting the constituents that follow. Thus, our initial grammar can be modified to handle relativization simply by adding the XG rules:

(6') relative --> relative-marker, sentence

(13) relative-marker ... direct-object --> relative-pronoun.

(14) relative-pronoun --> [that].

Figure 4 shows the XG derivation tree for "The man that John saw laughed".

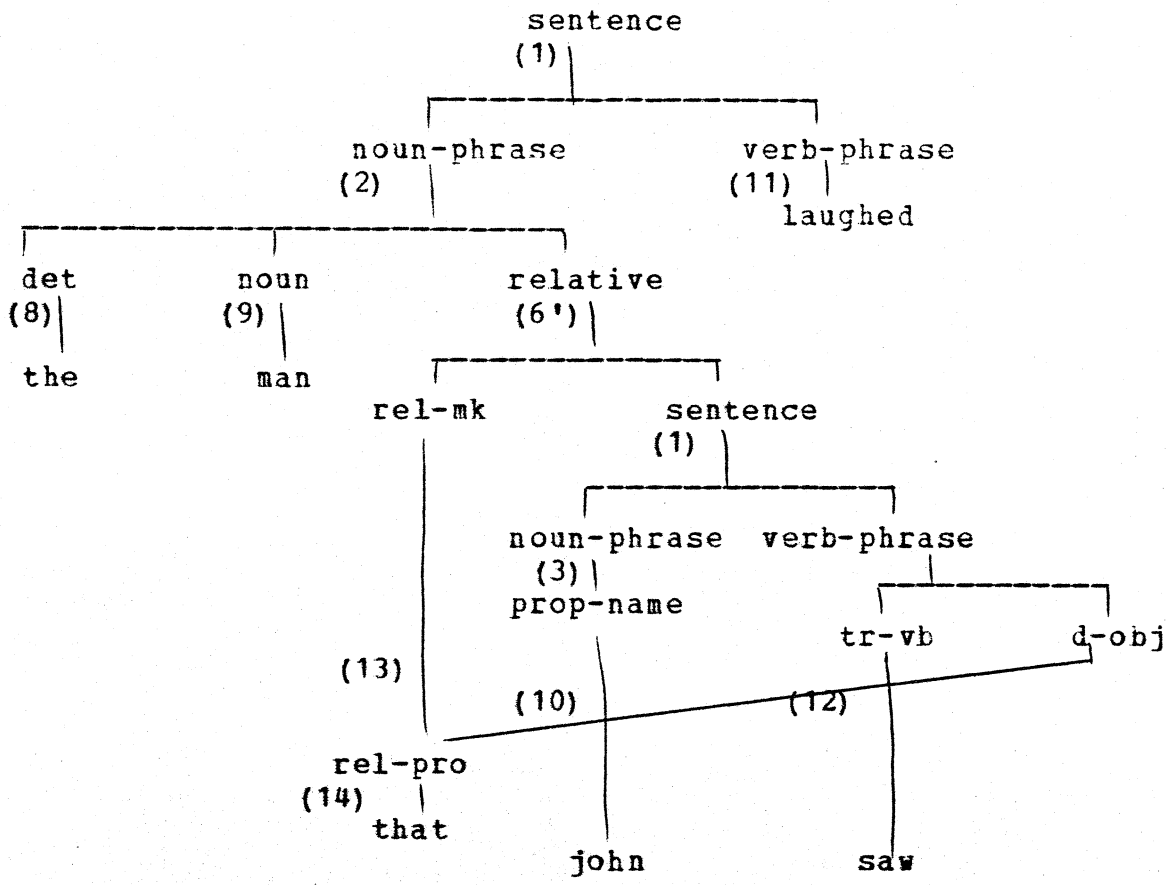


Figure 4. XG derivation tree for "The man that John saw laughed".

4.4. RGs

Restrictions can be used in RGs for the purpose of enforcing context sensitive constraints, but transformations seem to require an RG extension - possibly in the form of an additional component - , which is presently under study. (Hirschman & Puder, 1982)

An interesting feature of RGs is that a parse tree is automatically constructed during the parse (i.e., the tree-building parameters are hidden from the user). This makes a grammar clearer, but at the same time less flexible: only the history of rule applications is recorded, whereas in any other logic grammar the user may build up (through explicit parameters) any desired representation for the sentences parsed. The effects of context sensitivity, on the other hand, are ensured by giving each restriction access to the entire previously constructed parse tree. This need is the main difference between restrictions and the standard Prolog calls allowed in logic grammars (which are also, after all, procedure calls interspersed within the rules).

In short, where DCGs accommodate context-sensitive constraints within user-controlled parameters, RGs enforce them through restrictions placed upon a system-controlled parse tree. This concept would result in a higher level formalism if it gave the user a fairly complete independence from parse tree concerns. However, efficient exploitation of XGs requires some knowledge of

the parse tree, and the user needs to express restrictions in the lower level terms of tree traversal rather than in the typically declarative, operationally independent fashion of logic grammars.

5. How evolved a grammar formalism do we need?

Work in theoretical linguistics has lately been departing from transformational theory (Chomsky 1965), largely because of the subtelty of rules involved and the supplementary devices needed (e.g. co-indexing, filters, etc.) and because of the complexity of dealing with semantics within the transformational panadigm. Work by Montague (Montague 1976) and Gazdar (Gazdar 1981) resulted in a simpler and more intuitive formalization of semantics based upon the rule-to-rule hypothesis (Bach 1976): to each syntactic rule corresponds a structually analogous, semantic rule for building up logical representations.

Gazdar's framework, in particular, can deal with a wide range of syntactic phenomena within a phrase-structure theory that has a node admissibility interpretation rather than a generative one. This new outlook, augmented by metagrammatical devices (such as categories with gaps, metarules and rule-schemata) elegantly captures such important constructs as coordination and unbounded dependencies. Gazdar's "augmented phrase structure" approach has influenced research in AI, where the transformational approach had also been losing adepts, as it was also felt to deal insufficiently with semantics and, moreover, with sentence analysis - AI's main concern in natural language processing.

Among the systems inspired by this approach are (Joshi 1982, Robinson 1982, Schubert & Pelletier 1982).

Logic grammars, from all our previous discussion, would seem to provide an adequate computational framework within which to implement the augmented phrase structure approach. From the descriptive point of view, as we have seen, "logical" context-free rules are more powerful than standard ones because of parameters in grammar symbols, and unification. Procedure calls are moreover an inherent feature that is useful for representing constraints.

But, although any logic grammar supports at least this, the user need not be restricted to context-free type rules. MGs or XGs will moreover provide for generalized type-0 rules, and even for the handling of gaps, while maintaining high standards of efficiency.

Extra power available, therefore, can only represent a gain, since it does not preclude resorting to more elementary approaches as a special case. In this respect we support Berwick and Weinberg's contention that there is a possible tradeoff between parsing efficiency and descriptive apparatus, and that "a language that is quite 'high up' in the Chomsky hierarchy - e.g. a strictly context-sensitive language - may in fact be parsed more rapidly than languages lower down in the hierarchy - e.g. faster than some context-free languages - if the gain in succinctness is enough to offset the possible increase in parsing

time" (Berwick & Weinberg, 1982).

Our approach is therefore that of continuing research on logic grammar extensions that may be useful in view of a more powerful and elegant, while still efficient, treatment of some natural language processing phenomena. In the next section we describe a new logic grammar formalism developed in particular for dealing metagrammatically with coordination, but that exhibits several other features that are interesting by themselves.

6. Modifier structure grammars (MSGs)

Research by Michael McCord (McCord 1980, 198 has resulted in interesting ideas for processin natural language through logic grammars. In particular, the notion of modifier structure and the treatment of semantic interpretation presente there seemed a promising framework for solving nontrivial language-processing problems, such as coordination. Joint research with Michael McCord in view of a logic grammar, metagrammatical treatment of coordination, resulted in the development of a system consisting of: a) a new formalism for logic grammars, which we call modifier structure grammars (MSGs), useful for making modifier structure implicit in the grammar, b) an interpreter (or parser) for MSGs which also takes all the responsibility for the syntactic aspects of coordination, and c) a semantic interpretation component which produces logical forms from the output of the parser and deals with scoping problems, (as in McCord 81) which also includes specific rules for semantic interpretation of coordination. The whole system is implemented in Prolog-10 (Pereira, Pereira & Warren 1978). Here we make a brief presentation of this system. A complete description can be found in (Dahl & McCord, 1983).

MSG rules are of the form:

A : Sem --> B

where A --> B is an XG rule and Sem is a term called a semantic item, which plays a role in the semantic interpretation of a phrase analysed by application of the rule. The semantic item is (as in (McCord 1981)) of the form

Operator - LogicalForm

where, roughly, LogicalForm is the part of the logical form of the sentence contributed by the rule, and Operator determines the way in which this partial structure combines with others. Sem may be a "trivial" Sem if nothing is contributed.

When a sentence is analysed, a structural representation, in tree form, called "modifier structure" is automatically formed by the parser. Each of its nodes contains not only syntactic information but also the semantic information Sem supplied in the grammar, which determines the node's contribution to the logical form of the sentence (this contribution is for the node alone, and does not refer to the daughters of the node, as in Gazdar's approach (Gazdar, 1981)).

The semantic interpretation component first reshapes this tree into another MS tree where the scoping of quantifiers is closer to the intended semantic relations than to the (surface) syntactic ones. It then takes the reshaped tree and translates it into logical form. The modifiers actually do their work of modification in this second stage, through their semantic items. It should be noted that the addition of simple semantic

indicators within grammar rules contributes to maintain, from the user's point of view, a simple correspondence between syntax and semantics. This is similar in intention to the rule-by-rule hypothesis mentioned before (Bach 1976), but is differently realized: instead of a rule-to-rule correspondence, we have a correspondence between each non-trivial expansion of a non-terminal and a logical operator. That is, each time the parser expands a non-terminal symbol into a (non-empty) body, a logical operator labels the expansion and will be later used by the semantic component, interacting with other logical operators found in the parse tree obtained. The complexity of dealing with quantifier scoping and its interaction with coordination is screened away from the user.

With respect to coordination, the MSG grammar should not mention conjunction at all. The interpreter has a general facility for treating certain words as "demons" (cf. Winograd 1972), which trigger a backing up in the parser history that will help reconstruct ellisions and recognize the meaning of the coordinated sentence.

This proceeds in a manner similar to that of the SYSCONJ facility for augmented transition networks (Woods 1973, Bates 1978), except that, unlike SYSCONJ, it can also handle embedded coordination and interactions with extraposition. The use of modifier structures and the associated semantic interpretation component, moreover, permits in general a good treatment of

scoping problems involving coordination. Finally, the system seems reasonably efficient (cf. timings for our sample grammar in (Dahl and McCord, 1983)).

7. Concluding remarks.

Logic grammars, as we have seen, need not sacrifice efficiency to the goals of power and elegance. They seem to be evolving - like other computational formalisms - into higher level tools which allow the user to spare mechanizable efforts in order to concentrate on as yet unmechanizable, creative tasks.

We view MSGs as a step in that direction, with the main advantages of automatising the treatment of coordination, providing a modular treatment of semantics, and allowing the user not to worry over structure building.

The latter feature may be an attractive one for logic grammars in general to retain, since it makes a grammar easier to write and read, and more concise.

Since, moreover, logical structure desired for a sentence's final representation is also automatically built up, from a few simple semantic indicators in the grammar rules, it becomes easier to adapt a grammar to alternative domains of application: modifying the logical representation obtained need only involve the semantic components of each rule.

This modular isolation of structure lends grammars a

syntactico-semantic flavour. It may be viewed as a way out of the dilemma on whether the semantic component should be separate or intermingled with the syntactic one. Compromising on manipulating static semantic indicators during the syntactic parse while using them dynamically during the semantic one may well prove to be the way of combining the advantages of both approaches while minimizing the disadvantages.

ACKNOWLEDGEMENT

This work was completed under NSERC Operating Grant 12436 and PRG Grant 6-4240.

REFERENCES

Bach, E. (1976). An extension of classical transformational grammar. Mimeo, Univ. of Massachusetts, Amherst, MA.

Bates, M. (1978). The theory and practice of augmented transition networks. In: L. Bolc (ed.) Natural Language Communication with Computers. (Springer, Berlin, May 1978).

Berwick, R. C. and Weinberg, A. S. (1982). Parsing Efficiency, Computational Complexity, and the Evaluation of Grammatical Theories. Linguistic Inquiry, vol. 13, No. 2, Spring 1982, pp. 165-191.

Chomsky, N. (1965). Aspects of the Theory of Syntax MIT Press, Cambridge, Massachusetts.

Coelho, H. M. F. (1979). A program conversing in Portuguese providing a library service. Ph.D. Thesis, Univ. of Edinburgh.

Colmerauer, A. (1973). Les systemes-Q ou un formalisme pour analyser et synthetiser des phrases sur ordination. Publication interne No 43, Dept. d'Informatique, Universite de Montreal.

Colmerauer, A. (1975). Les grammaires de metamorphose. Groupe d'Intelligence Artificielle, Univ. de Marseille-Luminy. As "Metamorphosis Grammars" in: L. Bolc (ed.), Natural Language Communication with Computers (Springer, Berlin, May 1978).

Dahl, V. (1977). Un systeme deductif d'interrogation de banques de donnees en espagnol. These de Doctorat de Specialite, Univ. d'Aix-Marseille.

Dahl, V. (1981). Translating Spanish into logic through logic. American Journal of Computational Linguistics, Vol. 7, No. 3, pp. 149-164.

Dahl, V. (1982). On database systems development through logic. ACM Transactions on Database Systems, Vol. 7, No. 1, March 1982, pp. 102-123.

Dahl, V. and McCord, M. (1983). Treating coordination in logic grammars. Internal report, Univ. of Kentucky.

Gazdar, G. (1981). Unbounded dependencies and coordinate structure. Linguistic Inquiry, Vol. 12, No. 2, Spring, 1981.

Hirshman, L. and Puder, K. (1982). Restriction grammar in Prolog. Proc. First International Logic Programming Conference, Marseille, pp.85-90.

Joshi, A. (1982). Phrase Structure Trees Bear More Fruit than You Would Have Thought. American Journal of Computational Linguistics. Vol.8, No.1, Jan-March 1982.

Kowalski, R. A. (1979). Logic for problem solving. North-Holland Elsevier, New York, 1979.

McCord, M. (1980). Using slots and modifiers in logic grammars for natural language. In: Artificial Intelligence.

McCord, M. (1981). Focalizers, the scoping problem, and semantic interpretation rules in logic grammars. University of Kentucky.

Montague, (1974). English as a formal language. In: Thomason, R. H. (ed.), Formal Philosophy: selected papers of Richard Montague. Yale Univ. Press, New Haven, CT., pp.188-221.

Pasero, R. (1982). A dialogue in natural language. Proc. First International Logic Programming Conference, Marseille, France, pp.231-239.

Pereira, F. (to appear). Extraposition grammars. To be published in: American Journal of computational linguistics.

Pereira, F. and Warren, D. (1980). Definite clause grammars for natural language analysis - a survey of the formalism and a comparison with augmented transition networks. Artifical Intelligence 13 (1980), pp.231-278.

Pereira, F. and Warren, D. (1981). An efficient easily adaptable system for interpreting natural language queries. Dept. of AI, Univ. of Edinburgh.

Pereira, L., Pereira, F. and Warren, D. (1978). User's guide to DECsystem-10 Prolog. Div. de Informatica, LNEC, Lisbon and Dept. of AI, Univ. of Edinburgh.

Pereira, L. M. et al. (1982). ORBI - An expert system for environmental resource evaluation through natural language. Universidade Nova de Lisboa.

Robinson, J. A. (1965). A machine-oriented logic based on the resolution principle. J. ACM 12, pp.25-41.

Robinson, J. (1982). Diagram: A grammar for dialogues. Comm. ACM, January 1982, Vol.25, No.1.

Sabatier, P. (1980). Dialogues en francais avec un ordinateur. Groupe d'Intelligence Artificielle, Univ. d'Aix-Marseille.

Silva, G. M. T. et al. (1979). A knowledge-based automated message understanding methodology for an advanced indications system. Rome Air Development Center, Report RADC-TR-79-133.

Simmons, R. F. and Chester, D. (1979). Relating sentences and semantic networks with clausal logic. Dept. of Computer Science, Univ. of Texas.

Schubert, L. and Pelletier, F. (1982). From English to Logic: Context-Free Computation of "Conventional" Logical Translation. American Journal of Computational Linguistics, Vol.8, No.1, Jan-March 1982.

Winograd, T. (1972). Understanding natural language. Cognitive Psychology, 3, pp.90-93.

Woods, W. A. (1970). Transition network grammars for natural language analysis. C. ACM 13.

Woods, W. A., Kaplan, R. M. and Nash-Webber, B. (1972). The lunar sciences natural language information system: final report, BBN Report 2378.

LOGIC DATA BASES VS DEDUCTIVE DATA BASES

Working Paper to be presented at the

Logic Programming Workshop 1983

ALBUFEIRA - PORTUGAL

Hervé GALLAIRE
Laboratoires de MARCOUSSIS
Centre de Recherches de la **C.G.E.**
Route de Nozay
91460 MARCOUSSIS - FRANCE

ABSTRACT

The area of data bases is the area of Computer science most likely to be invested by a new methodology -should one say a new technology- based on logic programming. This survey investigates various approaches to the merging of these two worlds, trying to straighten out the advantages, problems and applications of each of them.

INTRODUCTION

The area of data bases is one more area of computer science subject to being taken over by a new methodology -should one say a new technology- based on logic programming. This paper surveys the various approaches to merging these two fields, depending on viewpoints adopted for one's problem analysis ; the logic data base field starts from logic and tries to enhance it with data base assets, be they data access techniques or data base features ; on the converse deductive data bases are built from existing data base systems by enhancing them with deductive, and other, capabilities. These two viewpoints, although yielding different systems and being interesting for different types of applications and goals, are rather complementary and share many common problems. The paper concludes that enough of the theoretical aspects of the deal are well-known and that it is time now for practical applications as well as theoretical improvements.

The paper is divided into five sections. The first section presents the four approaches to linking data bases and logic ; the first three to be described in sections 2 through 4 adopt the logic viewpoint and culminate into full blown logic database ; section 5 presents the deductive database approach.

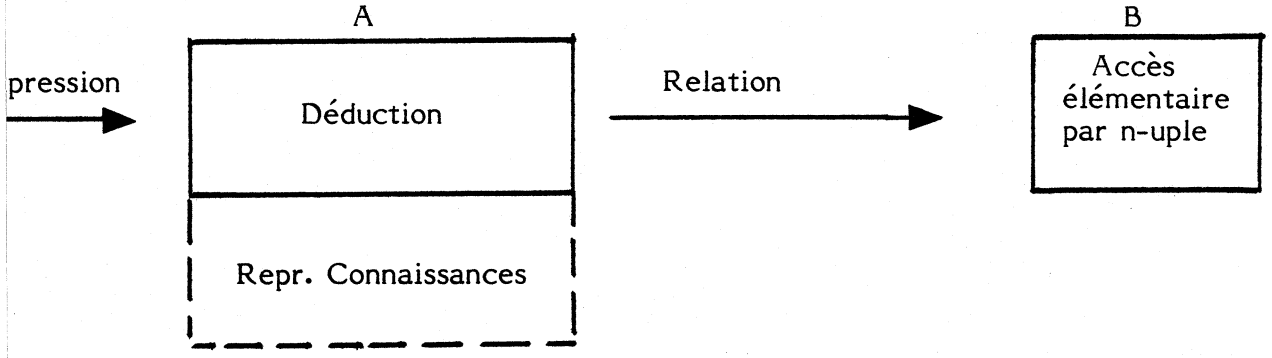
One should take note that the modeling power of logic databases will not be discussed in this overview because the paper does not deal at all with work on knowledge representation formalisms.

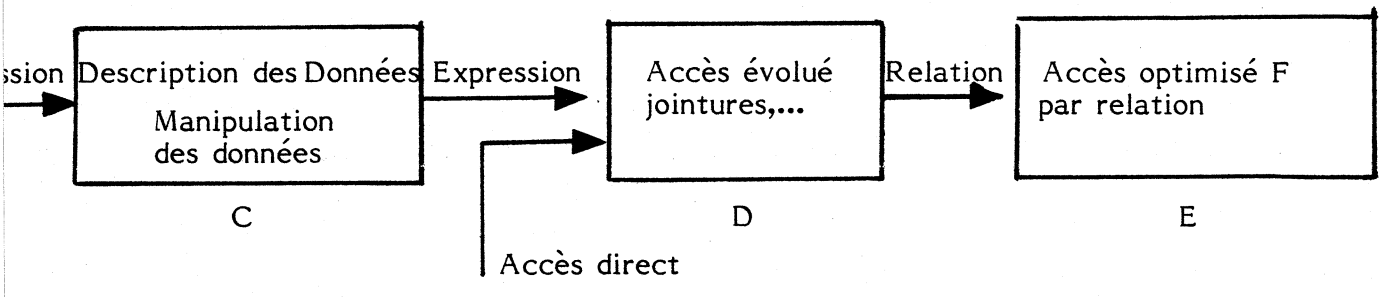
Section 1 : Logic programming - Data Bases

The first to realize the potential of logic programming for data bases was probably C.GREEN (1) who, although he did not know about logic programming which did not exist at that time, described how to connect logic-based Question-Answering system to data Base systems. Since that time various papers, books and workshops dedicated to that subject have brought up the subject (2, 3, 4, 5, 6) without fully clarifying the relationships between the two fields.

In order to study this relationship closely, we first decompose a logic programming system and a data base system into their respective components. A logic programming system, PROLOG being the most well known example of them, is made of a deductive component (A) and of a rudimentary access component (B) which provides the deductive component with individual tuples ; the query to A may be a relational expresssion (usually a negative clause in PROLOG) ; the interface between A and B is a relation.

A database system is made of a data description and data manipulation component (C), a data access expression optimizer (D), and a data access component (E) ; query relational expressions are submitted to (C) or to (D) ; interface between (C) and (D) is a relational expression, usually of the relational algebra ; interface between (D) and (E) is at the relation level, bringing back full sets of tuples instead of individual tuples as (B).





With such decompositions in mind, four types of connections can easily be thought of :

PROLOG+ : Some relations are defined as being managed by a mechanism of type E, thus giving a system made of :

$A \Leftrightarrow (B+E)$

PROLOGDB : PROLOG formulas can be considered as being a full fledged query language for a database access system (D-E) ; thus we obtain :

$A \Leftrightarrow (D+E)$

Logic Data Base : This is the natural extension of the previous two approaches where one builds above or aside PROLOG a true data base system, with a description and manipulation language, including capabilities for integrity constraints expressions etc... Although it is not necessary to include capabilities of type D or E they will be included if only for performance reasons ; thus we obtain :

$C \Leftrightarrow A \Leftrightarrow (D+E)$

with $C \Leftrightarrow A$ as a minimum system.

Deductive Database : The goal here is to provide extensions to conventional database systems which have well-known limitations, if only for the query languages which need to be embedded into foreign programming languages. The systems so obtained are of the type :

$A \Leftrightarrow C \Leftrightarrow D \Leftrightarrow E$

or even $C' \Leftrightarrow A \Leftrightarrow C \Leftrightarrow D \Leftrightarrow E$

when one combines this deductive database approach with the logic database one. Logic covers various aspects that the query language covers inadequately: views, optimizing techniques, theoretical understanding of important problems such as incomplete information handling,...

Section 2 : PROLOG+

It is known that PROLOG-like access to individual data is not well-suited to relations which would be stored in secondary memory, due to the fact that data is requested one at a time. Also it is known that even in primary memory there are ways to index data which make it faster to retrieve (e.g indexing on specific fields of a relation rather than sequential access on its name). On the contrary database systems are very much concerned with the efficiency of data retrieval. PROLOG+ systems are nothing more than systems in which some relations have been declared as database relations or DB-relations and handled by a DB-like access mechanism (indexing, B*-tree, multiple hashing,...). Such systems have already been built ; PROLOG-like access is simulated for the DB relations by buffering the set of tuples retrieved in one operation, and giving PROLOG one tuple at a time from this buffer. See eg (7). It is clear that this approach is an easy way to enhance PROLOG, and for some well defined large applications of PROLOG, worth implementing. It is surprising that no such a large scale application has been reported up to now.

Section 3 : PROLOG BD

Recall the configuration of such systems :

$A \Leftrightarrow (D+E)$

Such a system can be seen as a PROLOG+ system in which instead of interfacing with the DB system at the relation level, one interfaces at the resolvent level : given DB-relations, given other relations (called PROLOG relations or P-relations to distinguish them from DB-relations), given a PROLOG program including clauses mixing P-relations and DB-relations, one would like to optimize access to P-expressions i.e. to expressions containing P-relations only, rather than to evaluate each P-relation when, in the deductive part of PROLOG, it becomes the leftmost literal of the resolvent (as done in PROLOG+). There are several ways to do this which are examined below. First let see why one would want to do such global retrieval as opposed to an individual, relation-based retrieval ; among the possible reasons one which is most appealing is that it is known that DB systems behave more efficiently than virtual memory systems, that they have quite efficient optimizers, that they offer set-operators which can be very much optimized and even executed through specific hardware (the database machines).

The connection sketched above is in principle easy to imagine. A major initial decision to be made is how much control over the evaluation process is left to the programmer ; in other words the decision is to be made whether the programmer can decide (i.e can tell the system) when a (sub-) expression is to be sent to the database system, how much data is to be brought back, etc.

Making such a possibility explicit in the hands of the programmer requires an extension of the logic language, namely that a set of system predicates be added which allows to express information about retrieval, insertion, deletion, etc., thus making a "data sublanguage" out of PROLOG by extending it. Such an explicit control has been defined and advocated in (8) ; it could be a basis of some of the 5G languages. One could perharps also adapt to DB the technique of (9). These approaches are certainly worth experimenting, but we believe it is not easy : it is certainly not a simple matter to find logic programmers knowledgeable enough to make the right decisions about these retrieval expressions. Nevertheless it is the one which, in the short term, could prove the most effective ; one should bear in mind, though, that some DB researchers express concern about optimization problems and believe that DB access optimizing is a formidable task that needs much processing power, which is sometimes counter-intuitive, and which is usually better carried out by general programs.

If the responsability of the decision is to be taken by the system and not by the programmer, it remains two basic roads. The first is the compilation technique in which one translates an initial request into a DB-expression which is then sent to the DB-system ; thus there is a clear cut separation between deduction (generation of an evaluable expression) and access. The second technique is the interpretation one, in which both processes are intermixed.

Compilation

This technique has been widely studied (10) and has led to several implementations and approaches depending on the complexity of the logic program.

Case 1

There is no recursive axiom in the program for defining P-relations in terms of DB-relations.

This case is without difficulties. There are two ways to deal with it. One can modify the logic interpreter so that it delays evaluation of DB-relations until the resolvent involves DB-relations only ; this might perharps be done by using Geler (Freeze) predicate from PROLOG II. Alternatively one could write a translator which acts as a meta-interpreter as done in (11, 12).

Case 2

There exist recursive axioms in the program ; an example of such a case would be a transitive closure relation supposed to be a P-relation and defined in terms of itself (hence the recursivity) and a DB-relation. Whereas in case 1 all that was to be done was a macro-expansion, one is now confronted to a true program generation problem ; at least in principle two classes of solutions have been studied :

pseudo-compilation : This is an extension of case 1, i.e the recursive program is not translated into an iterative one, or into an evaluable formula ; rather it generates

a sequence of evaluable formulas each corresponding to an alternative solution used on backtracking when the logic interpreter, or the user asks for additional solutions. An example extracted from (7) follows :

given ancestor(X,Y) ← parent(X,Y)

ancestor(X,Y) ← ancestor(X,Z), ancestor(Z,Y)

and a query ← ancestor(X,Y)

the system will generate the following evaluable formulas :

[edb(parent,X,Y)] then

[edb(parent,X,Z), edb(parent,Z,Y)] then

[edb(parent,X,Z), edb(parent,Z,Z1), edb(parent,Z1,Y)]

...

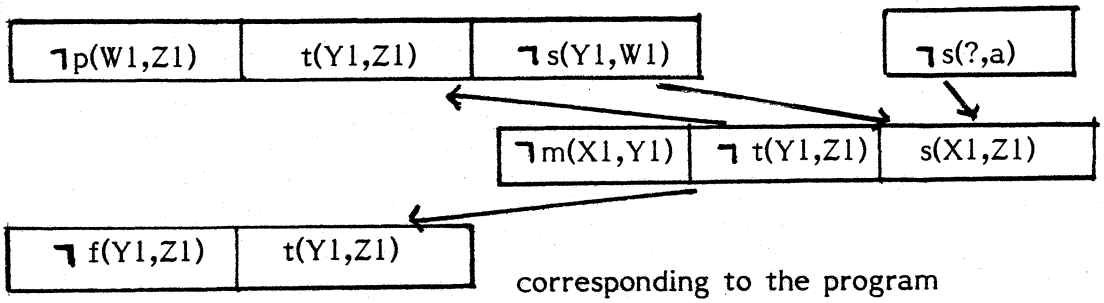
where edb(parent,-,-) is a relation evaluable by the DB ; such formulas evaluation can be optimized. Other examples show that additional capabilities (one should notice the example used a non-trivial recursion) such as negation and mixed relations can be handled too. A mixed relation is a relation defined by a program which includes assertions i.e positive litterals as well as conditionals (general axioms as above).

Although such systems are, in principle, simple enough, their drawback is a redundancy which is obvious from the example above : consecutive formulas share common literals which will be evaluated several times ; getting rid of this redundancy at the deductive system level amounts to a true compilation (see next) ; getting rid of it at the DB level is not a classical operation of such systems.

True compilation : It is possible to generate truly iterative programs involving purely evaluable DB-relations starting from recursive logic programs including both P-relations and DB-relations. Several techniques have been proposed (13, 14, 15).

In (13) recursive programs of the regular type (in the formal language sense) only can be handled ; it is not surprising that such a class of programs can be translated into iterative programs, as this is well known from automata theory. In (14) various extensions to the regular programs are given, without reaching the full power of logic programs.

(15) describes the most general approach as of to-day, it is based on connection graphs, a well-known technique (16) ; the basic idea is to generate a program which is a loop around the cycle(s) in the connection graph, collecting all DB-relations involved in this process until the exit of the loop. A simple example is in order (15) : given the following connection graph and a query s(?,a).



corresponding to the program

s(X1,Z1) ← m(X1,Y1), t(Y1,Z1)

t(Y1,Z1) ← s(Y1,W1), p(W1,Z1)

t(Y1,Z1) ← f(Y1,Z1)

with p,m,f DB-relations, the program to be generated goes along the loop collecting p-tuples, each of them driving an inner evaluation loop of m-tuples and f-tuples as can be seen by looking at the successive evaluable formulas :

m(?,Y1), f(Y1,a)

m(?,Y2), m(Y2,Y1), f(Y1,W2), p(W2,a) (r2)

m(?,Y3), m(Y3,Y2), m(Y2,Y1), f(Y1,W2), p(W2,W3), p(W3,a) (r3)

....

The program is :

```
Z1 = a
edb(p,W2,Z1) ; edb(m,X,Y1) ; edb(f,Y1,Z1) ; print(X)
enqueue(Q,W2)    values of W2 will drive an outer loop
foo = m(X2,Y2), m(Y2,Y1), f(Y1,W2)   to be evaluated, starting from
                                      f for each value of W2
i = 2
while (Q ≠ empty) do
   while (Q1≠empty) do W2=Deque(Q1) ; edb(foo) ; print(Xi) ; od
     does what was expected, see(r2) above
   Q1 = Q

   Q = empty
   while (Q1 ≠ empty) do W3 = Deque(Q1) ; edb(p(W2,W3) ;
                                enqueue(Q,W2) ; od
     collects now values for W2 as in r3 above
   replace m(Xi,Yi) by m(X_{i+1},Y_{i+1}), m(Y_{i+1},Yi) in foo ; i=i+1 ;
                       prepare for a new outer loop
od
```

This program is, on the surface, satisfactory ; the authors state that its only limitation is due to the fact that the form of the initial query must be known (here s(?,a)). There may be another difficulty which is that, in order for the program to stop, the enqueue operation is not a mere "push" : it must check that the value has not been pushed i.e. enqueued before ; this may be a practical limitation of the system.

Another approach, without any of these limitations maybe under way (17) but not enough is known about it at this time.

Interpretation

These techniques intermix deduction and evaluation steps ; in fact what was described in section 2 for PROLOG+ was already an interpretation. Other schemes have been presented, starting from the idea that unification done tuple at a time was not precisely adapted to systems in which DB-relations were handled ; such a case was argued in MRPPS (4) where the concept of Π-unification was developed. A more systematic study in terms of PROLOG implementation is described in (11) where the basic idea is the following : rather than storing at each node of the proof tree the whole set of unifications (as a table), it is possible either to store a unification set only at the root and to store at each node the computation rules which will allow to compute their new unification sets from their parent node, or to store unification sets at the leaves and at each node the information which allows to compute their unification set from their descendent nodes. Examples are described in (11) although a complete implementation of PROLOG based on this has not been realized.

Such techniques would be interesting for parallel PROLOG implementations.

Section 4 : Logic DB

This approach is the most natural one for all those who believe logic programming to be a universal programming language. Their arguments are strong, we adhere to them basically. A database, as seen by KOWALSKI (6,18) is a collection of HORN clauses including functions if one wishesto (already an extension of conventional DB), atop of which it suffices to build DB functionalities.

A starting point is that PROLOG, including its set-of extension is relationally complete, i.e. can express all queries expressed in relational algebra, the common base language to all relational DB systems (with operators such as union, projection, join, division,...) ; such a result although interesting is well-known since CODD results on equivalence between relational calculus (i.e. logic) and relational algebra. Of course the set-of construct gives all that is needed to express aggregation constructs, averages,...

However, PROLOG and logic provide more than a conventional query language because the expressive power of logic programming is at least that of least fixed points, an example of which being transitive closures :

lfp(R,R*) can be expressed as a simple PROLOG program computing the least fixed point R* for any PROLOG relation R. In specific cases, it is simpler to compute the closure directly, as in

ancestor(X,Y) ← parent(X,Y)

ancestor(X,Y) ← parent(X,Z), ancestor(Z,Y)

Some limitations of the approach should nevertheless be phrased :

- It must clearly be connected to a DB system as described in Section 3 if only for efficiency problems ; this is clear for example in (19) where a set of queries to a DB system expressed in PROLOG had to be optimized before being sent to the DB system ; although one could argue that one of the major difficulties (duplicates) in the answers came from the PROLOG evaluation scheme itself, not from logic, this is still a problem to be faced in general.

- HORN clauses, if relationally complete, are not sufficient to express naturally all queries that one would like to ask using logic itself (6,18) : find all suppliers supplying all pieces needed for project "a".

  Such a query involves conditionals within conditionals ; this is translated into negation within the body of a clause and is not properly handled by PROLOG unless specific attention is paid.

- Integrity constraints, time-constraints involves additional mechanisms which resemble plan-generation techniques ; non-monotonic reasoning is also necessary ; possible solutions are presented in (6,18,20).

Some realizations have been reported along these lines, eg (21,22). The first one is a PROLOG implementation of QBE, while the other is a description of a system where PROLOG is an intermediate language target for a QBE external language as well as an SQL external language and a relational algebra external language. In the PROLOG implementation of QBE (21) it is shown how to simply take into account integrity constraints on inserts and deletes using a technique which was also used in (23), the catchall clause. That logic database approach is typically an approach which is closest to Artificial Intelligence, at least to the theorem proving part of Artificial Intelligence if not to the knowledge representation one. Systems built in that perspective include (4, 24). Powerful non-HORN theorem provers can be used, plan-generation techniques can be expressed.

Section 5 : Deductive Data Bases

The bias introduced in developing deductive database systems is that DB systems can be enhanced by adding to conventional retrieval capabilities of data explicitly introduced, that of retrieval through deduction mechanisms using general laws. This extension, introduced at first purely for retrieval purposes turns out to have many more facets which are briefly examined.

Conventional DB's manipulate facts only (the tuples of the relations). The general laws they use are so-called integrity constraints (IC), used to validate updates of facts. All queries are evaluated with respect to facts only.

In deductive DB's general laws can be partitioned in two sets : IC's and deductive rules (DR). Queries are then evaluated with respect to facts and DR's. But IC's will also need to be evaluated with respect to facts and DR's. This makes it more difficult of course to check IC's which thus require deductive capabilities. Deductive databases (DDB's) are made of a collection of solutions to various problems whose conventional solutions in DB's have to be adapted in this new context. To understand these new solutions, old problems and solutions must first be reviewed.

Conventional DB's enforce implicit assumptions for retrieval :

- Closed world assumption (3, 36) : all facts not known to be true, i.e not stored as tuples, are false

  $\neg R(a1,...,an)$ iff $< a1,...,an > \notin R$

- Unique names : elements with different names are different

  $\forall\, b,c \qquad b \neq c$

- domain closure : there are no other elements than those stored in the DB.

The first two hypotheses combined allow negation evaluation (recall that NOT is an operator in relational algebra). The third one allows evaluation of queries such as $\forall\, xP(x)$,... It could be dispensed of if one restricted the allowable queries to meaningful subsets of the syntactically correct queries, thus reducing to range-restricted queries.

$\forall x(Q(x) \rightarrow P(x))$ is evaluable without hypothesis(3)

while $\forall\, xP(x)$ is not.

These conventional assumptions have to have counterparts in any formalized view of conventional DB's. After this formalization is done, it is possible to extend it to DDB's. Two formal views of conventional DB's have been studied (25, 5) : a model-theoretic view (MTV) and a proof-theoretic (PTV) one. Without going into details, the MTV assumes that the set of facts is an interpretation E, a model, of a theory made of IC's and that query evaluation is done in E, abiding to the above three assumptions. Although such a view deals with problems such as query evaluation and optimization,

choice of conceptual schemas, etc... it does generalize to DDB's and incomplete information problems. The PTV sees a conventional DB as a first-order theory T plus a set of closed formulas, the IC's. The theory T is made of facts (positive HORN formulas) and a set of particularization axioms. These particularization axioms (Domain closure, Uniqueness of names, completion, equality) are the formal translation of the above three assumptions. The DB is still not a DDB but deduction could be used to handle T ; this may be unwise and in any implementation this is likely to be dealt with at a metalevel, i.e integrated to the query algorithm. Nevertheless PTV is very useful in terms of the generalizations it suggests :

- DDB's which are obtained via a third class of axioms, the deductive rules (DR) mentioned earlier.

- DB's which allow disjunctive information, leading to incomplete information (5, 26, 27).

DDB's are subject to new problems, in that the axioms introduced in T may be inconsistent with some general deductive laws ; it is well known that such is the case between disjunctive axioms and those (in T) accounting for CWA.

$\neg R(a1,...,an)$ iff $\{T, DR\} \vdash R(a1,...,an)$

cannot be accepted as such :

$Cat(X) \rightarrow Black(X) \cup White(X)$ (DR)

$Cat(Felix) \leftarrow$

axioms in T

$\vdash$ Black(felix) hence $\neg$ Black(Felix)

$\vdash$ White(felix) hence $\neg$ White(Felix)

These two informations are contradictory with the unique DR. Solutions to handle this are partially known (5, 26, 27) and consist either in restricting general laws (DR) to regular clauses with adequate axioms T' instead of T, or in dealing with incomplete information systems.

It must be emphasized again that this theoretical view (regular clauses + axioms T') is not to be implemented as such ; again, implementation goes through some meta-rules rather than using T' axioms ; for instance negation as failure (33) and range-restricted formulas (35).

There are two ways to exploit a DDB. Most of the systems realized today use the deductive approach where data is actually deduced when needed. In the generative approach (28), deductive rules are used as generative rules : each time data is entered, all information derivable from it, or with its help, is derived and generated (stored in the DB) ; of course supressing data becomes a non-trivial process, akin to Truth Maintenance Systems in AI since generation is similar to forward system in AI. The generation task appears to be prohibitive in terms of computation overhead, but it may not be so depending on the context of application.

Finally, one should note that DDB's are not yet fully understood ; however they already permit various generalizations of conventional DB's among which generalized notions of views, integrity constraints, query languages, data dependencies studies, etc (29, 30, 31, 32,...). Obviously, not all of these notions have an acceptable treatment : among them one can mention update of views, recursive DR's, checking IC's, etc.

It should be clear from the above discussion how close are some of the problems which are dealt with both from the DB viewpoint and from the logical one ; what to emphasize and how to solve problems, is where these two fields separate.

CONCLUSION

In this overview paper, two main trends for enhancing data bases on one side, logic on the other, have been examined. Both aim at bridging the gap between DB and logic. One puts the emphasis on efficiency, the other on functionalities. As a result there is no single logic & DB system : a taxonomy of systems including DB's, knowledge-based systems, logic interpreters handling large sets of assertions, etc can be developed ; corresponding to this taxonomy which is rather intuitive and well-known, another one has been proposed here according to the emphasis on logic or on DB's : PROLOG+, PROLOGDB, logic DB, deductive DB. Yet, another taxonomy is still to be developed : it has to do with the types of axioms that could be sufficient for the purpose of each type of system corresponding to the above taxonomies. As an example, consider recursive axioms : what is the complexity of such axioms when one adopts the deductive DB perspective ? Isn't it sufficient to have the power of transitive closure ? Then, isn't it possible to take advantage of such a simplification in the deductive system to be built. Such questions are important and the task of finding such a taxonomy is now to be undertaken. It may be presently undertaken in the framework of the Japanese 5G Project which aims at the same objective : bring together logic and database system. One should note that we have not covered the use of logic as an implementation language for interfacing DB's, e.g. for a natural language interface (19) or for menus and other tools (37). Finally recall that an important topic has not been discussed here at all : the knowledge representation problem and the contribution of logic databases to it.

ACKNOWLEDGMENTS

Views expressed here result from many years of studies in common with J.M. NICOLAS, and also discussions with J. MINKER ; our collaboration started with the logic and databases publication and is still continuing. The influence of R. KOWALSKI and of R. REITER's work should be obvious throughout.

REFERENCES

(1) **GREEN C.**, "Theorem proving by resolution as a basis for Question-Answering systems", Machine Intelligence 4 (MELTZER, B., and MICHIE, D., eds), American Elsevier Pub. Co., NEW-YORK (1969), pp. 137-147

(2) **GALLAIRE H., MINKER J.**, eds, "Logic and Data Bases Plenum Press, NEW-YORK (1978).

(3) **NICOLAS J.M.** and **SYRE J.C.**, "Natural question-answering and automatic deduction in the system SYNTEX", Proc. IFIP 74, North-Holland, AMSTERDAM (1974), pp. 595-599

(4) **MINKER J.**, "An Experimental relational database system based on logic" in (2), pp. 107-147

(5) **REITER R.**, "Towards a logical reconstruction of relational database theory", unpublished manuscript

(6) **KOWALSKI R.A.**, "Logic as a database language", Proc. advanced seminar on TIDB, Cetraro (Sept. 1981)

(7) **BRUYNOOGHE M.**, "PROLOG-C implementation", University of LOUVAIN, 1981

(8) **MIYAZAKI N.**, "A data sublanguage approach to inferfacing predicate logic and relational databases", ICOT report, 1982

(9) **CLARK K.L.** and **Mc CABE F.**, "The Control facilities of IC-PROLOG", In "Expert Systems in the Micro Electronic Age" (Ed. MICHIE), EDINBURGH University Press, 1979

(10) **GALLAIRE H., MINKER J.** and **NICOLAS J.M.**, "An overview and introduction to logic and databases", in (2).

(11) **CHAKRAVARTY U.S., MINKER J.** and **TRAN D.**, "Interfacing predicate logic languages and relational databases", Proc. lst Int. Conf. on logic programming, MARSEILLE (Sept. 1982), pp. 91-98

(12) **KUNIFUJI S.** and **YOKOTA H.**, "PROLOG and relational databases for fifth generation computer systems", ICOT Report presented at CERT 82 workshop "Logical Bases for Databases".

(13) **CHANG C.L.**, "DEDUCE 2 : further investigations of deduction in relational databases", in (2), pp. 201-236

(14) **MINKER J.** and **NICOLAS J.M.**, "On recursive axioms in deductive databases", Information Systems 7,4 (1982)

(15) **HENSCHEN L.** and **NAQVI S.**, "Compiling recursive databases", submitted to JACM (1982)

(16) **SICKEL S.**, "A search technique for clause interconnectivity graphs", IEEE Transactions on computers, Vol. C-25, n° 8, 1976

(17) **NAQVI S., FISHMAN D.** and **HENSCHEN L.J.**, "An Improved compiling technique for first-order databases", Presented at CERT 82 Workshop "Logical Bases for Databases", Bell laboratories and Northwestern University

(18) **KOWALSKI R.**, "Logic Programming", Invited paper IFIP PARIS Sept. 19-23

(19) **WARREN D.H.D.**, "Efficient processing of interactive relational database queries expressed in logic", Proc. 7th VLDB Conf., CANNES (Sept.1981), pp. 272-281

(20) **BOWEN K.A.** and **KOWALSKI R.A.**, "Amalgamating language and metalanguage in logic programming", in "Logic programming" (K.l CLARK and S.A. TARNLUND eds), Academic Press, LONDON (1982), pp. 153-172

(21) **NEVES J.C., ANDERSON S.O.** and **WILLIAM H.**, "A PROLOG implementation of Query-by-Example", Proceedings 7th Int. Computing Symposium, March 22-24, 1983, NURNBERG

(22) **LI D.Y.** and **HEATH F.G.**, "ILEX : an intelligent relational database system", HERIOT-WATT University, Dept. of Electrical and Electronic Engineering, EDINBURGH 1982

(23) **GRUMBACH A.**, "Knowledge Acquisition in PROLOG", 1st Int. Logic Programming Conf., Sept. 14-17th, MARSEILLE

(24) **KELLOGG C.** and **TRAVIS L.**, "Reasoning with data in a deductively augmented data management system", in Advances in Database Theory, Vol.1 (Plenum 1981), pp.261-295 (H.GALLAIRE, J. MINKER, J.M. NICOLAS editors)

(25) **NICOLAS J.M.** and **GALLAIRE H.**, "Database : theory vs. interpretation", in (2), pp. 33-54

(26) **BOSSU G.** and **SIEGEL P.**, "La saturation au secours de la non monotonicité", Thèse de 3è cycle, Université de MARSEILLE-LUMINY, MARSEILLE (Jun.-1981), to appear in A.I.

(27) **MINKER J.**, "On indefinite databases and the closed world assumption", Proc. 6th Conf. on Automated Deduction, in Lecture Notes in Computer Science, Vol. 138, Springer-Verlag, NEW-YORK (1982)

(28) **NICOLAS J.M.** and **YAZDANIAN K.**, "An outline of BDGEN : a deductive DBMS", Techn. Rep., ONERA-CERT, TOULOUSE (Oct. 1982)

(29) **NICOLAS J.M.** and **YAZDANIAN K.**, "Integrity checking in deductive databases", in (2), pp. 325-344

(30) **BLAUSTEIN B.T.**, "Enforcing database assertions : techniques and applications", Ph.D. Thesis, HARVARD Univ., CAMBRIDGE (Aug. 1981)

(31) **PIROTTE A.**, "High level database query languages", in (2), pp. 409-436

(32) **FAGIN R.**, "HORN Clauses and databases dependencies", J.ACM 29,4 (Oct. 1982), pp. 952-985

(33) **CLARK K.L.**, "Negation as failure", in (2), pp. 293-322

(34) **GALLAIRE H., MINKER J.** and **NICOLAS J.M.**, "Logic and Databases - An overview and survey", Joint report CERT-CGE-Univ. of MARYLAND

(35) **DEMOLOMBE R.**, "Utilisation du calcul des prédicats comme langage d'interrogation des bases de données", Thèse de doctorat d'Etat, ONERA-CERT, TOULOUSE, Feb. 1982

(36) **REITER R.**, "On closed world databases", in (2), pp. 55-76

(37) **PEREIRA L., FIGUEIRO M :** Relational Databases à la carte, Centro de Informatica, Universidade Nova de Lisboa, PORTUGAL

Computing with Sequences

C.D.S.Moss. Feb 1933

Abstract

All Prolog implementations deal implicitly with sequences of solutions to problems by means of backtracking: if one solution to a subproblem is rejected another is presented. A number of implementations also provide predicates which provide sets or bags (sets with possibly repeated elements) of solutions. But in these case the system finds all the solutions before proceeding. It is suggested here that an implementation of bags, or sequences, which finds only one solution at a time can be integrated easily with existing implementation techniques for Prolog.

This technique has a number of advantages over similar proposals. If it is combined with subprograms which exhibit tail recursion, then one can write logically correct programs to process sets of solutions which do not use extra memory. These can include programs which reduce solutions (e.g. count, sum or average solutions) without using "impure" techniques. However the technique involves little overhead in programs which do not use it, unlike certain other proposals for coroutining.

This will have particular advantage in database applications where large amounts of information must be retrieved serially from secondary storage. It also has potential application for parallelism since the next solution of the subproblem may be pre-evaluated on a parallel processor, without changing the normal interpretation of Prolog clauses.

Introduction

The normal mode of evaluation in Prolog may be termed a "semi-lazy" evaluation of all the solutions of a goal. In other words, one solution is computed to a subproblem, and the computation is then suspended until another solution is required. This produces the very attractive "stack" discipline which characterises Prolog and contributes significantly to its speed and memory efficiency.

But in many cases one wishes to talk about "all" solutions to a problem. A facility is provided in several Prolog implementations (e.g. Warren [1982]) by an evaluable predicate which produces tne solutions as a list:

The predicate "setof(A,B,C)" means that C is a list of the variables A which solve the problem B.

In many situations, particularly if one wishes to nest calls to such a predicate, it is desirable to provide the solutions as a set, in which any duplicate solutions have been removed. But this clearly entails extra work, and presents questions if some of the solutions contain uninstantiated variables (does one want the most general or most specific answers?). Hence some implementations also provide a "bagof" predicate which is defined in the same way, but can contain duplicate answers. But this is also implemented by producing all solutions to the problem at one time and therefore involves allocating (implicitly) enough space to hold all the solutions.

Because of this, many programmers will persist in using the impure "hacks" that were common in Prolog before these predicates were introduced. These involve making temporary assertions in the database to hold information which is not lost on backtracking. Apart from the loss of speed from using these techniques, programs can become obscure, as the technique effectively introduces global variables into a language which avoids their use otherwise. In the context of parallelism, such usage is doubly suspect.

Using Sequences

The introduction of sequences has two parts: an evaluable predicate and a "lazy" way of processing (only) lists. We will introduce a new predicate called "seqof". It has the same definition as "setof" or "bagof" except that solutions produced in the list may be repeated, and the order of solutions is defined by the program. i.e.

seqof(A,B,C) means that the list of all solutions for A in goal B is a list C.

Let us demonstrate the use of this by showing a procedure which computes the average population of all countries in a database.

```
averagepop(A) <- seqof(B, (population(C,D), country(C)), E),
        average(E,0,0,A).

average(nil,0,0,0).
average(nil,A,B,A/B).
average(A.B,C,D,E) <- average(B, C+A, D+1, E).
```

Here "averagepop" computes the average population and "average" is a general procedure for computing averages, working in a "bottom-up" fashion so that tail recursion is applicable. Note that infix function calls to arithmetic predicates are assumed. A definition of all predicates used may be found at the end.

The meaning of this program may be appreciated quite separately from the method of implementation: "seqof" generates a list containing the populations of every country in the database, and "average" acts recursively on this list to produce the average.

However the implementation suggested is as follows: seqof(A,B,C) evaluates B until a single solution is found, when it binds C to A.x (where x is defined below), or if no solution is found then C is bound to nil. Control then passes to the subgoals following seqof - in this case "average" (unless C is already bound fully to a list in which case the next solution of B is found). Execution proceeds normally until an attempt is made to bind a non-variable to the lazy list object, the value called "x" above. If this is normal list-processing, then it will be an attempt to bind it to some term "D.E", or "nil". At this point, control is returned to "seqof" and another solution is attempted. If it is found, then the value D will be instantiated and processing returns to the list consuming procedure.

One valuable aspect of this control mechanism is that the "seqof" procedure can be programmed so that when control returns to it, it can detect whether any valid references still exist to the first solution (because of backtracking points etc), and if not, can delete that solution from the stack. In this way, the global stack space used can be reclaimed and a list of all solutions used without consuming an equivalent amount of stack space.

A Database Example

As a further example of the use of this approach, consider another database query. Buneman et al [1982] consider queries such as:

"find the names of employees who are under 30 years of age and are paid more than the average salary for all all employees."

In a typical database query language this might be expressed:

```
retrieve NAME from EMPLOYEE where AGE<30 and
   SALARY > average(retrieve SALARY from EMPLOYEE).
```

They demonstrate that this can be expressed in a purely functional query language by an expression

```
!EMPLOYEE o |([[AGE,30]o LT, [SAL,AVESAL] o GT] o AND) o *NAME;
```

where "o" represents function composition, AVESAL is a function which computes the average salary, "!" generates a stream of all employees, "|(" restricts this stream by the following predicate, and "*" converts a stream of values into a

character stream.

A system such as Chat-80 (Warren, Pereira 1979) might represent this as a Prolog query in the form:

```
ans(ANS) <= seqof(A, employee(B) & age(B,C) & C<30
                  & salary(B,D) & D>E &name(B,A), ANS)
          & seqof(F, employee(G) & salary(G,F), H)
          & average(H,0,0,E).
```

While this (relational) form of the enquiry is not as concise as the functional form it has an important advantage. The variable "E" which represents the average salary is independent of the expression in which it appears and it is clearly not necessary to compute it for each employee. In the Chat-80 system the query would be rearranged and constraints inserted to ensure that it is only calculated once (Warren 1981). In a functional system, the optimiser has to recognize a "constant subexpression".

The processing of the query clearly involves two passes over the employee relation, which may be presumed to be large and stored in a database. When computing the average salary there is only a need to retain the partially computed average. The second pass generates the names of employees, which will presumably be printed out and also not stored.

Buneman et al point out that there is a further possible optimisation for this query: if there are no employees less than 30 it is unnecessary to compute the average salary. The implementation of sequences proposed here would not easily allow for this.

Implementation Details

This discussion has ignored the vital implementation details of how the stack is organized to handle several different "control points" simultaneously (and one must remember that seqof may be called at several points in direct or indirect recursion). There are in fact a number of possible implementations and which is used may depend on several factors.

1. One can implement full "spaghetti stacks" in which several computations can interleave. This is the approach of Clark and McCabe [1979]. This does not unfortunately mesh very well with existing implementation techniques.

2. One can start a new stack for the seqof predicate separate from the old stack. In fact it is possible to consider this as a completely separate "process", as will be explored below, and in a virtual memory environment where address space is no problem (though real memory is) this may well be the best solution.

3. One can allocate (by some heuristic) a certain space on the stack for the seqof process in addition to that consumed by the first solution and start the following processes above this. On return to the interrupted process, it performs a "context switch" which resets the top of available stack space to the predefined space. If this space is filled, it is then necessary to "stack-shift" the rest of the stack to make space. Fortunately this is an operation which is already done by several implementations to allow for the several different storage areas used by Prolog.

The advantage of methods 2 and 3 (and there may be better methods) is that they do not incur a large time penalty on normal execution. The convenient and efficient stack handling of Prolog is preserved and the normal sequencing of goals is only interrupted when treating a "lazy" list.

It is however necessary to modify the unification routine to recognize a new object - the lazy list. However this does not affect the binding variables to the object, only the attempt to match it with another list object (list functor or nil). Hence the modification is limited to the case of binding a functor to a functor - and even in compiled code this is normally performed by a subroutine. It is to be hoped that with the gradual introduction of microcoding for unification this will not be a significant problem and be outweighed by the saving in space achieved.

Some other uses of this technique

This technique opens up new possibilities for Prolog evaluation, of which three will be explored briefly.

1. Input-Output in Prolog programs is generally handled in a manner which is both semantically impure and obscure from a programming point of view. Consider the following programming fragment found in several systems:

```
get(X) :- repeat, get0(X), X = ' ', !.

repeat.

repeat :- repeat.
```

Here the predicate "get0" unifies the next character in the input (of some unspecified file) with the parameter X. If it is not a space, then the system backtracks to get0, which is not implemented as a procedure with backtrack points. However backtracking to "repeat" always succeeds and the next character of the input is read. Thus "get" reads the next non-blank character from the input. However, both "get0" and "get" are predicates which have side-effects. Backtracking over the input file does not "rewind" the file as one might expect.

Let us suppose that input-output predicates were defined in terms of the seqof predicate. Thus, for instance, a call to the predicate file(A,B) binds B to a character in file A. A call to seqof(B,file(A,B),C) binds C to the list of characters in file A. If this is the predicate used then correct backtracking behavior will be acheived automatically with more economical use of storage. Of course a more sophisticated implementation might yield even greater economy.

2. Consider the following use of seqof:

seqof(A, proc1(B,A), C), seqof(D, proc2(C,D), B).

Here proc1 and proc2 are two processes which each "consume" a list which is their first parameter and produce answers which are their second parameter. If the first parameter is considered a list of input messages, then the behavior of these two processes may considered to be one of message passing from one to the other. Initially proc1 is invoked and produces a message A. Then proc2 is invoked and produces an answer D. When it tries to consume the second message, control is passed back to proc1. Notice that there does not need to be any one-to-one correspondence between the number of messages provided by proc1 and proc2. Also deadlock is easily detected.

3. It is possible to replace any subgoal in a Prolog program by a pair of goals which are equivalent:

goal(A,..,Z) to

seqof(A:..:Z, goal(A,..,Z), C), soln(C,A:..:Z)

where A:..:Z represents a tuple of the variables appearing in the goal and soln is defined conceptually by

soln(A.B, A).

soln(A.B, C) :- soln(B, C).

Thus we may consider any subgoal in a Prolog program as a separate subprocess which generates a sequence of solutions which is then passed on to its neighbors. It is possible to implement these in the most convenient manner: one possibility that is attractive is for a subgoal to produce exactly one more solution than has yet been demanded by the other goals. Then when the next subgoal backtracks there will be a solution immediately available. However the demand for new processes will not grow exponentially as could be the case if indefinite parallelism were allowed.

Conclusion

The introduction of the seqof predicate and the "partially evaluated list" technique has introduced into Prolog a very limited form of coroutining. It has the advantage of preserving

the behavior of programs written for a left-to-right depth first evaluator and avoiding the overhead which has accompanied many other proposals for coroutining. It has the disadvantage of being somewhat "delicate" in its space-saving value: if the user leaves other references to the list at some point after the seqof call, then it may not be possible to discard the intermediate solutions.

Definition of predicates used

In the following definitions, the name of the predicate is given first followed by the number of its parameters; then an English definition of the predicate in which variables A,B,C etc. are used to represent the 1st, 2nd, 3rd etc. parameters of the predicate respectively.

bagof/3 -- the bag of all solutions for A in goal B is a list C.

seqof/3 -- the sequence of all solutions for A in goal B is a list C.

setof/3 -- the set of all solutions for A in goal B is a list C.

averagepop/1 -- the average population of all countries is A.

average/4 -- the average value of sublist A of a list, having a partial total of B from C items in the head of the list, is D.

population/2 -- the population of A is B.

country/1 -- A is a country.

soln/2 -- B is a member of the sequence A.

employee/1 -- A is the identification of an employee

age/2 --y the age of employee A is B

salary/2 -- the salary of employee A is B

name/2 -- the name of employee A is B

References

P. Buneman, R.E. Frankel, R. Nikhil [1982]: An Implementation Technique for Database Query Languages. ACM Trans. on Database Systems. 7/2, pp164-186.

K.L. Clark, F. McCabe [1979]: The Control Facilities of

IC-Prolog. In D. Michie (ed): Expert Systems in the Microelectronic age. E.U.P.

D.H.D. Warren, F.C.N. Pereira [1978]: An efficient easily adaptable system for interpreting natural language queries. D.A.I. Paper 155, Univ. of Edinburgh.

D.H.D. Warren [1981]: Efficient Processing of Interactive Relational Database queries expressed in logic. VLDB Conf. pp272-281.

D.H.D. Warren [1982]: Higher order extensions to Prolog: are they needed? Machine Intelligence 10, pp441-454.