LOGIC PROGRAMMING NEWSLETTER 4

WINTER 1982/1983

CONTENTS

EDITOR

Prof. Luís Moniz Pereira
Departamento de Informática
Universidade Nova de Lisboa

PUBLISHED

UNIVERSIDADE NOVA DE LISBOA
Centro de Informática
Faculdade de Ciências e Tecnologia
2825 Monte da Caparica — Portugal
☎ 245 4214 · 245 4987 · 245 4464 · 245 5299
245 5642 · 245 5643 · 245 5644 · 245 5645

ACKNOWLEDGEMENT

This publication has been fully subsidized by Junta Nacional de Investigação Científica e Tecnológica (JNICT), Av. D. Carlos I, 126, Lisboa, Portugal.

COLABORATION

Thanks are due to Alexandre Silva and António Porto for help with the address list.

SUBMISSIONS

Send to the editor, typed or printed, if possible double spaced, one side of the paper only.

CONTRIBUTIONS

All those who can contribute please do so, by sending at least US$6.00 or equivalent, per number received, to the editor, payable to the Centro de Informática da Universidade Nova de Lisboa.

DESIGN, TYPESETTING AND PRINTING

Serviços Gráficos da Universidade
Nova de Lisboa
Av. Miguel Bombarda, 20-1.º
1000 Lisboa, Portugal

EDITOR'S FOREWORD

● *Last September, in Marseille, took place The First International Logic Programming Conference. Some 150 people were present from twenty countries: Argentina, Australia, Belgium, Brazil, Canada, Czechoslovakia, Denmark, France, Germany, Hungary, Israel, Italy, Japan, Norway, Poland, Portugal, Sweden, United Kingdom, USA and Yugoslavia. The proceedings, containing the 35 accepted papers, can be obtained with a check for US$30 plus postage payable to ADDP — Association pour la Diffusion et le Développement de Prolog, to be sent to*

ADDP — GIA
Case 901
Faculté des Sciences de Luminy
13288 Marseille Cedex 9
France

● *The Second International Logic Programming Conference will take place in Uppsala, Sweden, in 1984. For more information contact the program chairman*

Sten-Åke Taïnlund
Dept. of Computing
Uppsala University
Sturegaten 4A
75223 Uppsala, Sweden

● *A Logic Programming Workshop will be held in the south of Portugal, in the last week of June (27 Jun-1 Jul) 1983. For further information contact*

Luís Moniz Pereira
Dept. de Informática
Universidade Nova de Lisboa
Quinta da Torre
2825 Monte da Caparica, Portugal

● *To receive the Newsletter more often simply contribute more often. More contributions are needed in the way of local news, personal news, contracts, grants, visits, posts available, and other interesting community news items, such as new available software, efficiency comparisons of implementations, etc. Also, short communications are welcome. Last but not least, monetary contributions are indispensable.*

ON APPLYING AN INDUCTIONLESS TECHNIQUE TO PROVE PROPERTIES OF RESTRICTED PROLOG PROGRAMS

R. Barbuti, P. Degano, G. Levi

Istituto di Scienze dell'Informazione
Universitá di Pisa
Corso Italia, 40 I-5600 PISA ITALY

The aim of this note is to show an application of an inductionless technique for proving properties of logic programs to an example. Our method applies to annotated Horn clauses, where the inputs and the outputs are explicitly distinguished, as introduced by [1, 2]. This restriction, while maintaining most of the expressive power of PROLOG, allows to define multi-output functions. Furthermore, in such a framework, we can adopt a proof technique originally developed in single-output functional approach, i.e. the Knuth-Bendix completion algorithm for term rewriting systems [3, 4]. Roughly speaking, such a technique consists in showing that a set (composed of equations, with a single output, defining a set of procedures, and the property to be proved) results in a consistent set of definitions. In order to prove consistency no induction is required, yet it is sufficient to show that, given a term, it cannot be reduced by means of the equations and the property to two different terms.

In order to naturally express properties, we extend PROLOG clauses to have more than one literal in their left-hand side. Accordingly, the way a goal is computed has been modified, so that properties could be applied to, say, speed up the computation. In order to apply to annotated PROLOG our extension of the Knuth-Bendix completion algorithm, some restrictions are to be imposed on PROLOG. The first restriction is in connection with the distinction between inputs and outputs, and rules out the possibility of inverting a function. The second allows to write deterministic procedures only, and the others concern well-formedness of definitions. A detailed description of our extended/restricted PROLOG can be found in [5], as well as a complete presentation of our method.

A program is a set of clauses $\{H_i \leftarrow\!- R_i\}$, H_i *being an atom and* R_i *a set of atoms, such that whenever* H_j *overlaps* H_k *with substitution* ϑ, *then* R_j *overlaps* R_k *with* ϑ. *A property is a clause* $P_1, \ldots, P_n \leftarrow\!- Q$, *such that* P_i *overlaps some* H_j. *A program is consistent either if it does not contain any property, or if it contains some properties which preserve determinism of the results of computations.*

Given a consistent program F *and a property* P, *our algorithm, if succeeding, shows that the new program* F' *obtained by adding* P *to* F *is consistent too. In other words, property* P *has been shown to be a theorem in the theory defined by program* F.

We assume familiarity with the Knuth-Bendix completion algorithm as presented for instance in [3].

Consistency is checked by showing that the new program F' *is relational confluent. Relational confluence means that, given any goal, all its computations do terminate with the same result. The method we use for proving that a clause* P *is a theorem of a consistent program* F, *consists in determining all those (symbolic) goals to which both* P *and some clause in* F *can be applied. All possible computations of such goals are performed (symbolically evaluating the goals without instantiating their input variables), then their results are checked for (symbolic) equality.*

Let us show by means of an example how we prove a property of a program.

Let us have the following clauses, where x, y, z, t, w, r *denote variables over binary digits, and* α, β, γ, δ, σ, ρ *denote binary numbers, i.e. sequences of binary digits built by applying the constructor "." to a sequence and a binary digit, and starting from the empty sequence* λ.

R1. Bplus(in: x,x,z; out: z,x) <−−
R2. Bplus(in: x,y,x; out: y,x) <−−
R3. Bplus(in: x,y,y; out: x,y) <−−
R4. Cplus(in: λ,0; out: λ) <−−
R5. Cplus(in: λ,1; out: λ.1) <−−
R6. Cplus(in: α.x,y; out: γ.t) <−−
Cplus(in: α,z; out: γ), Bplus(in: x,y,0; out: t,z)

R7. Nplus(in: α, λ; out: α) <−−
R8. Nplus(in: λ, β; out: β) <−−
R9. Nplus(in: α.x, β.y; out: γ.t) <−−
Nplus(in: α, δ; out: γ), Cplus(in: β,z; out: δ),
Bplus(in: x,y,0; out: t,z)

Clauses R1-R3 *define the standard 3-adic (addend1, addend2, carry) binary addition whose value is the pair (result, carry). Clauses* R4-R6 *define the sum of a bit over a binary number (carry over). Clauses* R7-R9 *define the sum of two binary numbers.*

Let us prove the following trivial property

P. Nplus(in: α.0, β.y; out: γ.y) <−− Nplus(in: α,β; out: γ)

We look for the most general symbolic goals to which P *and some* R_i *are applicable. Only* R9 *overlaps* P *yielding the following goal.*

G1. <−− Nplus(in: α.0, β.y; out: γ)

Note that the output variables are left unbound.
By applying P *to* G1 *we obtain the binding* γ = δ.γ *and*

L1. <−− Nplus(in: α,β; out: δ)

By applying R9 *and* R2 *the bindings* γ = σ.t *and* t=y, z=0 *are found and* G1 *evaluates to*

L2. <−− Nplus(in: α,ρ; out: σ), Cplus(in: β,0); out: ρ)

Since the bindings introduced during the two symbolic evaluations are compatible, we set L2 <−− L1 *as a new property to be proved. With the following property*

P1. Nplus(in: α,ρ; out: δ), Cplus(in: β,0; out: ρ) <−−
Nplus(in: α,β; out: δ)

the property P *is trivially proved by reducing* L2 *to* L1.
Note that P1 *has more than one literal in the left-hand side, connected by variable* ρ.

Now, we look for all the most general goals that can be obtained from P1 *and* R1-R9, *namely superpositioning* Nplus(in: α,ρ; out: δ) *with* R7-R9 *and* Cplus(in: β,0; out: ρ) *with* R4-R6.
No most general goal can be generated from P1 *and* R7 *(nor* R9), *since*

binding ρ *with the empty number* λ *(with* β.y*) would cause the output of* Cplus *to be bound to* λ *(to* β.Y*), too (recall that our programs cannot be inverted).*

From P1 *and* R8 *we obtain the goal*

G2. <-- Nplus(in: λ,ρ; out: γ), Cplus(in: β,0; out: ρ)

whose computations originate the new property

P2. Cplus(in: β,0; out: β) <--

From P1 *and* R4 *we generate the following most general goal to which both clauses can be applied.*

G3. <-- Nplus(in: α,ρ; out: δ), Cplus(in: λ,0; out: ρ)

whose computations lead both to the empty clause with compatible bindings.

Superpositioning P1 *and* R6 *yields the most general goal*

G4. <-- Nplus(in: α,ρ; out: δ), Cplus(in: β.x,0; out: ρ)

that evaluates in two different ways to the same literal, i.e.

Nplus(in: α, β.x; out: δ).

It remains to prove that property P2 *holds. The only most general goals are obtained from* P2 *and* R4 *and* R6*. The first goal is*

G5. <-- Cplus(in: λ,0); out: δ)

that is reduced to the empty clause by two different computations. The same situation arises with the second goal

G6. <-- Cplus(in: β.x,0; out: δ)

No new property is generated, thus the property P *has been proved.*

The aim of this short communication is to show that our extension of the Knuth-Bendix algorithm can be profitably used to prove logic program properties. Thus, we have not mentioned at all interesting problems such as those related to new property generation (e.g. L1 <-- L2 *or* L2 <-- L1 ?*), simplification of properties, termination of computations, and termination of the algorithm itself.*

REFERENCES

[1] BELLIA, M., DEGANO, P. and LEVI, G.: "The call by name semantics of a clause language with functions". In: Logic Programming, K. L. Clark and S-A Tarnlund Eds., Academic Press, 1982.

[2] CLARK, K. L. and GREGORY, S.: "A relational language for parallel programming". Proc. Functional Programming Languages and Computer Architecture, Portsmouth 1981, 171-178.

[3] KNUTH, D. and BENDIX, P.: "Simple word problems in universal algebras". In: Computational problems in abstract algebra, J. Leech Ed., Pergamon Press 1970, 263-297.

[4] HUET, G. and OPPEN, D. C.: "Equations and rewrite rules: A survey". In: Formal Languages: Perspectives and Open Problems, Book R. Ed., Academic Press (1980).

[5] BARBUTI, R., DEGANO, P. and LEVI, G.: "Towards and inductionless technique for proving properties of logic programs". Proc. 1st Logic Programming Conference, Luminy, 1982.

A PROLOG IMPLEMENTATION OF SASL

Harvey Abramson

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

I have recently completed a Prolog implementation of SASL (St. Andrews Static Language), an elegant functional (applicative) language characterized by: definition by recursion equations; non-strict semantics of function application; lazy evaluation; weak typing [*Turner, 1976, 1979, 1981*]. The definite Clause Grammar formalism ([*Colmerauer, 1978*] *and* [*Pereira & Warren, 1980*]*) and a few associated Prolog predicates are used, following Turner, to "compile" SASL expressions to a string of combinators and global names. This string is evaluated, however, by a* normal order *reduction machine (also implemented in Prolog), rather than by Turner's normal graph reduction machine: a normal graph reduction machine is feasible in Prolog, but apparently only at the cost of modifying the Prolog data base once for each cycle of the reduction machine.*

Additions to SASL's global environment are made by the Prolog predicate define *(using the Waterloo Prolog notation):*

```
<- define("fac 0 = 1; fac n = n * fac(n-1)").
```

This results in the following clause of the predicate global *being added to the Prolog data base:*

```
global(fac,ap(ap(TRY,ap(ap(MATCH,constant(num(0))),num(1))),
    ap(ap(S,MULT),ap(ap(B,id(fac)),ap(ap(C,SUB),num(1))))))).
```

The second component of global *is the compiled version of* fac *with all local names removed:* ap(x,y) *denotes application of* x *to* y*, a primitive operation of SASL; the capitalized names are combinators defined in* [*Turner, 1981*].

In application, such as:

```
<-sasl("fac (suc n) WHERE n = 2, suc n = 1 + n",*Ans).
```

the SASL expression is compiled to:

```
ap(ap(U,ap(B,id(fac))),
    ap(Y,ap(U,ap(K,ap(K,cons(ap(ADD,num(1)),num(2)))))))).
```

The reduction machine is brought into action, and *Ans *is eventually instantiated to 6, the value of the expression.* "cons(x,y)" *denotes the pairing of* x *and* y *into a list, and is, like* "ap"*, a primitive operation of SASL.*

This implementation of SASL will be used, first of all, to implement and test the unification-based conditional binding constructs introduced in [*Abramson, 1982*].

It will also be used to test the feasibility of using SASL as a syntactic sugaring for Prolog predicates which are in fact functions of n *arguments, ie, predicates of* (n+1)*-arity with the first* n *as "input" arguments, and the* (n+1)*-st as the "output" argument. A possible use of this sugaring of functions would be to write:*

N is <sasl expression>

analogous to

N is <integer expression>

of DEC-10 Prolog, but allowing the definition of SASL functions and the manipulation of lists, strings, etc. If this sugaring turns out to be useful, SASL expressions then ought to be translated directly into Prolog rather than to a string of combinators which are then reduced.

I found, by the way, the use of Definite Clause Grammars and Prolog to be an elegant, almost ideal tool for language implementation. I had previously implemented another version of SASL in BCPL, using a lazy SECD machine to evalute expressions. The Prolog implementation seemed to me to require at least an order of magnitude less effort on my part to complete.

The interpreter is available on request.

REFERENCES

[ABRAMSON, 1982]
Abramson, "Unification-based conditional binding constructs", Proceedings-First International Logic Programming Conference, Université de Marseille, Luminy, 1982.

[COLMERAUER, 1978]
Colmerauer, A., "Metamorphosis Grammars", in Natural Language Communication with Computers, Lecture Notes in Computer Science 63, Springer, 1978.

[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.

[TURNER, 1976]
Turner, D. A. "SASL language manual", Dept. 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, 1979.

[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.

ACKNOWLEDGMENT: This work was supported by the National Science and Engineering Research Council of Canada.

A PROPOSAL

Masoud Yazdani

Department of Computer Science
University of Exeter, UK

In Prologs based on the Edinburgh syntax there is a rather strange way of distinguishing between assertions to the data-base and queries of the data-base.

i.e. if using a file then

<no prompt>⟶ is-father-of(John,Jane).
represents a statement of fact and

<user-typed prompt>⟶ *is-father-of(John,Jane).
represents a query.

However, if the user is at a terminal,

<system-typed prompt>⟶ *is-father-of(John,Jane).
represents a query, while the statement of fact is

<system-typed prompt>⟶ *assert(is-father-of(John,Jane)).

Why not improve the syntax by following (in our view) a more consistent convention both at file level and user-interaction level.

is-father-of(John,Jane).
to represent a statement of fact

is-father-of(John,Jane)?
to represent a query.

Micro-Prolog has attemped to solve this problem, with partial success, by introducing

Does(is-father-of(John,Jane)).
as query

Add(is-father-of(John,Jane)).
as statement of fact.

The result of the above modifications is the ommission of 'does' *and* 'add' *from micro-Prolog in favour of a simple convention on* '.', '?' *too.*

SYSTEM SIMULATION ON PROLOG BASIS

Iván Futó

Inst. for Coordination of Computer Techniques / SzKI /
Budapest, H-1368 POB. 224, HUNGARY

Tamás Gergely

Research Inst. for Appl. Comp. Sci. / SZÁMALK /
Budapest, H-1536 POB. 227, HUNGARY

To support discrete system simulation we elaborated an extended version of T-PROLOG [2], TS-PROLOG [3]. *In* TS-PROLOG *it is possible to define systems, system components, system and component activities, communication paths, activity interrupts and times /durations/ locales to the components. These are ensured by the use of predefined clauses.*

A TS-PROLOG *model consists of:* i/ *Taxonomy definition.* ii/ *Communication definition.* iii/ *Local time definition.* iv/ *Definition of interrupts.* v/ *Goal definition.* vi/ *Input form definition.* vii/ *Output form definition.* viii/ *Initialisation.* ix/ *Activity definition sections.*

For i-iv *and* vi-ix TS-PROLOG *provides built-in clauses to help the user.*

```
i/ System(S_name, S_activity, Start_s, End_s):
        Component(C_name, C_activity, C_start, C_end),
        :
        Elementary_component(E_name, E_activity, E_start, E_end).
   Components_of(C_name, _level):
        Component(C_name_1, C_activity_1, C_start_1, C_end_1),
        :
        Elementary_component(E_name_1, E_activity_1, E_start_1,
                             E_end_1).
```

System(N,A,St,Et), Component(N,A,St,Et), Elementary_component(N, A,St,Et) *define respectively a system, a compound component, an elementary component of name* N, *activity* A, *begining its activity at* St */expressed in local time/ and prescribed termination time* Et */in local time/. Activity is syntactically an atom which will be executed by a process* [2] *corresponding to the system element, as a procedure call.*

ii/ Communication(C_i, C_j, A_{ij}).
Communication(C_i, C_j, A_{ij}): Restrictions_for(A_{ij}).

The above clauses define an allowed communication by message sending from component C_i *to component* C_j *and the message is to be of form* A_{ij}.

iii/ Local time(C_i, R_i).

The above unit clause defines the local time / unit time ratio. C_i *is the name of the component,* R_i *is of form* T, .T, . .T *which means that respectively* T, 1/10 T, 1/100 T *local time unit is equivalent to one abstract time unit.*

iv/ Allowed_interrupt(C_i, C_j, A_j).
Allowed_interrupt(C_i, C_j, A_j): Side_effects.

The above definitions allows the process corresponding to component C_i *to interrupt the process corresponding to* C_j *and force it to execute procedure call* A_j */with* Side_effects/.

vi/ Input(A).
Input(A): Restrictions_for(A).

The form of assertions giving the input of the model are defined by the above clauses. When the TS-PROLOG *system initializes the model it verifies that all inputs were initialized in the prescribed form.*

vii/ Output(A).
Output(A): Restrictions_for(A).

The arguments and functor name of all unit clauses of form A *are automatically printed out by the* TS-PROLOG *system /simulated data/. Naturally it is possible to use the standard printing built-in predicates too, during the simulation process.*

viii/ *In the initialisation section the input assertions specified in section* vi *are given with concrete values for their parameters.*

ix/ *The activity definition section provides the "working part" of the model. Special built-in predicates are at the user's disposal to define component's activities. Some of them will be mentioned /in addition all built-in procedures of the* T-PROLOG *and* MPROLOG [1] *languages may be used/:* Sending(M), Sending(M, T) */T is the "delivery time of the message/,* Holding(T) */to suspend the process execution for* T *local time units/,* Interrupt(C_j), Waiting(M), Deletion(C_j), Introduction(C_j L, A, St, Et) */to delete and introduce components during program execution and modifying dynamically the original model,* Delete_communication (C_i, C_j, A_{ij}), Add_communication(C_i, C_j, A_{ij}) */to change dynamically communication paths/.*

TS-PROLOG *uses the* PDSS *subsystem* [4] *of the modified* MPROLOG [1] *system /*TM-PROLOG*/.*

Now we give a simple example to show a TS-PROLOG *program.*

*A system /*Space_system*/ consists of a space ship, an astronaut and of a control center. The control center defines a goal /a cosmic object/ to the astronaut and if it is acceptable, and reachable within the "life-time" of the astronaut then it will be visited.*

```
# Taxonomy definition #

System(Space_problem, No_activity, 0, 100):
        Component(Space_ship, No_activity, 0, 10),
        Elementary_component(Control_center, Goal_generation, 0, 100).
Components_of(Space_ship, 1--1): # level of spaceship 1--1 #
        Elementary_component(Astronaut, Decision_making, 0, 10).

# Communications #

Communication(Control_center, Astronaut, A_goal(_name,
        _access_time)): Identifier(_name), Number(_access_time).
Communication(Astronaut, Control_center, Acceptable(Yes)).

# Local time definitions #

Local_time(Space_ship, .5).
Local_time(Astronaut, .5).
Local_time(Control_center, .1).

# The time on board the space ship passes 5 times faster then on the
control center #

# Input definition #

Input(Acceptable_goal(_a_goal)): Identifier(_a_goal).

# Output definition #

Output(Visited(_cosmic_object, _time)).

# Goal definition #

Possible_goal(Alpha, 15).
Possible_goal(Beta, 5).
Possible_goal(Gamma, 5).

# Initialisation #

Acceptable_goal(Alpha).
Acceptable_goal(Beta).

# Activities #

Goal-generation:
        Possible_goal(_goal, _access_time),
        Sending(A_goal(_goal, _access_time)),
        Waiting(Accepted(Yes)).
Decision_making:
        Waiting(A_goal(_goal, _access_time)),
        Acceptable(_goal),
        Sending(Acceptable(Yes)),
        Holding(_access_time),  Local_time(_t1),
        Assclause(Visited(_goal, _t1)),
```

The solution is that Gamma is visited at Space_ship */Astronaut local time 5. /Alpha is not a solution because* 15 > 10 *the life time of the Astronaut, when Beta is not an acceptable goal. /*

REFERENCES

[1] BENDL, J., KÖVES, P. and SZEREDI, P.: "The MPROLOG system", Preprints of Logic Programming Workshop, Debrecen, Hungary, pp. 201-210, 1980.

[2] FUTÓ, I. and SZEREDI, J.: "T-PROLOG User Manual", Institute for Coordination of Computer Techniques 1981, Budapest.

[3] FUTÓ, I. and GERGELY, T.: "A Logical Approach to Simulation, to appear in the proceedings of the 'International Conference on Model Realism'", Bad Honef, FRG, 20-24.04.1982.

[4] KÖVES, P. anad SZEREDI, P.: A Programming Support Environment for PROLOG program development, presented at the Workshop "Logic programming for Intelligent Systems", Los Angeles, California, 18-21 aug. 1981.

A PROLOG DEMAND DRIVEN COMPUTATION INTERPRETER

Luís Moniz Pereira

Departamento de Informática
Universidade Nova de Lisboa
Quinta da Torre
2825 Monte da Caparica
PORTUGAL

This interpreter realizes the demand driven computation process described in [1]

Notes:

Predicate "stream" accepts a functional predicate goal R *and delivers a stream* X *of results, where difference lists are used to represent streams. After each value in the stream is produced it pauses, and displays the stream. If a* <CR> *is given it continues, if a* <space> *is given it shows the calls waiting to be demand driven and continues. Example call:* stream(p=:X).

*Predicate "*up_to*" produces up to* N *values of a stream for a given call. Example call:* up_to(3,conc([1,2],[3,4])=:X).

Predicate "#" evaluates any predicate call. If the predicate is functionally defined, it evaluates it recursively until a list is produced, where the head of the list, if any, contains the first result of evaluating the call, and the tail a call to a functional predicate for producing the next result. The call [] *evaluates to the empty list.*

To do so, it uses predicate "@", which picks up a clause for a functionally defined predicate and evaluates its body if there is one. However, if any argument is demand driven and is not yet evaluated, no clause can be picked up and "@" will evaluate the demand driven arguments, and return to "#" the call with its arguments evaluated.

```
?- op(230,xfx, =:).      /*  functional relations    */
?- op(240,fx , @ ).      /*  access to program clauses    */
?- op(240,fx , #).       /*  evaluation    */
?- op(254,xfx,<-).       /*  functional relations' conditions    */

/*    USER INTERFACE    */

stream(R=:X) :- s(R,X-X).

s(R,X-Z) :- # R=:A , !, ( (A=[] ; A=[_|T], list(T) ), Z=A          ;
                           A=[V|T], Z=[V|Y], show(T,X), s(T,X-Y) ).

show(T,X) :- write(X), get0(C), ( C=32, write(T), skip(10), nl ; C=10 ), nl.

up_to(N,R=:[V|Y]) :- N>0, # R=:[V|S] , !, M is N-1, up_to(M,S=:Y).
up_to(_,_=:[]).

/*    INTERPRETER    */

# R=:S  :-  ( list(R), R=S ; @ R=:A , # A=:S ) .
# (A,B)  :-  # A , # B .
# G      :-  G .

list([]).
list([_|_]).

/*    access to non-unit and unit functional predicate clauses, regular
      Prolog clauses, and system predicates                          */

@ G :- (G<-C), # C.                 /*    non-unit clauses    */
@ G :- G.                           /* unit clauses, Prolog, and system */

/*  user specified info about demand driven arguments  */

@ conc(A,X) =: conc(EA,X)           :-  # A=:EA.
@ select(N,A) =: select(N,EA)       :-  # A=:EA.
@ sift(A) =: sift(EA)               :-  # A=:EA.
```

```
@ filter(X,A) =: filter(X,EA)          :- # A=:EA.
@ merge(A,B) =: merge(EA,EB)           :- # A=:EA, #B=:EB.
@ mul(A,B) =: mul(A,EB)                :- # B=:EB.

/*   PROGRAMS   */

/*   conc   */

conc([],X)   =: X.
conc([X|Y],U) =: [X|conc(Y,U)].

/*   bounded buffer   */

bounded_buffer(WS,RS) =: AS <- bmerge(WS,RS,0,S1), buffer(S1,U-U)=:AS.

bmerge([write(X)|WS],RS,I,[write(X)|AS])  :- I<5, K is I+1, bmerge(WS,RS,K,AS).
bmerge(WS,[read|RS],I,[read|AS])          :- I>0, K is I-1, bmerge(WS,RS,K,AS).
bmerge(_,[],_,[]).

buffer([write(X)|S],V-[X|W])  =: buffer(S,V-W).
buffer([read|S],[X|V]-W)      =: [X|buffer(S,V-W)].
buffer([],_-[])               =: [].

/*   infinite list of integers   */

intfrom2 =: inc(2).

inc(X) =: [X|inc(K)] <- K is X+1.

n_integers(N) =: Y <- intfrom2=:X, select(N,X)=:Y.

select(0,_)      =: [].
select(N,[X|Y]) =: [X|select(K,Y)] <- N>0, K is N-1.

/*   primes   */

primes =: sift(intfrom2).

sift([X|Y]) =: [X|sift(filter(X,Y))].

filter(X,[Y|Z]) =: [Y|filter(X,Z)] <- Y mod X=\=0.
filter(X,[Y|Z]) =: [Y|filter(X,Z)] <- Y mod X=:=0.

/*   quicksort   */

qs([])     =: [].
qs([X|Y]) =: conc(qs(Y1),[X|qs(Y2)]) <- part(X,Y,Y1,Y2).

part(X,[H|T],[H|S],R) :- H=<X, part(X,T,S,R).
part(X,[H|T],S,[H|R]) :- H>=X, part(X,T,S,R).
part(_,[],[],[]).

/*   cyclic network of agents   */

p=:Y <- merge(mul(2,[1|Y]), merge(mul(3,[1|Y]), mul(5,[1|Y]))) =:Y.

merge([X|Y],[U|V]) =: [X|merge(Y,[U|V])] <- X<U.
merge([X|Y],[U|V]) =: [U|merge([X|Y],V)] <- X>U.
merge([X|Y],[U|V]) =: [X|merge(Y,V)]     <- X=U.

mul(X,[Y|Z])  =: [W|mul(X,Z)] <- W is X*Y.
```

REFERENCE

[1] HANSSON, A.; HARIDI, S.; TÄRNLUND, S.-Å: "Properties of a Logic Programming Language" in "Logic Programming" (K. Clark and S.-Å. Tärnlund eds.), Academic Press 1982, and also report 8/81, Computing Science Dept., Uppsala University, Sweden.

short communications

METACONTROL OF PROCESS SYNCHORONISATION IN T-PROLOG

Iván Futó, János Szeredi

Inst. for Coordination of Computer Techniques / SzKI /
Budapest, H-1368 POB. 224, HUNGARY

In T-PROLOG *to the initial goal sequence:* $A_1, \ldots, A_n$ *of a* PROLOG *program correspond* n *processes executing the goals conceptually in parallel* [3] /: New(A_1), ... , New(A_n) /.

To help potential users, a very simple but powerful mechanism is provided for user's scheduling algorithm definition [2]. *To define such algorithm the full power of the* MPROLOG [1] *system can be used.*

This mechanism consists of a pair of built-in procedure calls, Suspend *and* Activate_suspended(_1st).

The execution of procedure call Suspend *causes the unconditional suspension of the currently active process. If there is no more activable process in the system the built-in scheduler of* T-PROLOG *executes an* Activate_suspended(_1st), *call where* _1st *is an input parameter which has the form of a list of process identifiers. This list must be defined by the user as an output of a* MPROLOG *program sequence. For ex. to define a* Last Suspended First Activated *scheduling algorithm we can do the following:*

```
: New(Run(Process_1)), New(Run(Process_2)), New(Run(Process_3)).
# The corresponding goal sequence in PROLOG: Run(Process_1),
  Run(Process_2), Run(Process_3). #
Run(_n):
        Suspension, Newline, Outterm(_n).
Suspension:
        Active_process(_ap),
          Assclause(To_be_activated(_ap),1),
            Suspend.
Activate_suspended(_1st):
        LSFA(_1st, 1).
LSFA(_ap._1st,_n):
        Getclause(To_be_activated(_ap) ,n), /,
          Delclause(To_be_activated,n), Plus(_n, 1, _n1),
            LSFA(_1st, _n1).
LSFA(Nil, _n).
```

Active_process(_ap), Assclause(_cl, _n), Getclause(_cl, _n) *and Delclause are built-in predicates giving the identifier/name of the currently active process /internal or user defined name/, introducing, getting and deleting clauses from a partition.*

The output of the above program is:

```
Process_3
Process_2
Process_1
```

REFERENCES

[1] BENDL, J., KŐVES, P. and SZEREDI, P.: "The MPROLOG system", Preprints of Logic Programming Workshop, Debrecen, Hungary, pp. 201-210, 1980.

[2] FUTÓ, I. and SZEREDI, J.: "T-PROLOG User Manual", Institute for Coordination of Computer Techniques, Budapest, 1981.

[3] FUTÓ, I. and SZEREDI, J.: "T-PROLOG: a very high level simulation system", Logic Programming Newsletter no. 2, Autumn 1981.

USING PROLOG TO ASSESS SECURITY RISKS IN DATA PROCESSING SYSTEMS

John M. Carroll, Oi-Lun WU

Computer Science Department
University of Western Ontario
London, CANADA

This work indicates that PROLOG can be used to analyse data-processing systems in accordance with the non-discretionary data-flow security model. The PROLOG language allows the analyst to introduce work-factor considerations so that outputs that have security implications can be identified and accorded the protection they deserve.

The requirements of the non-discretionary data-flow model of computer security can be summarized in two commandments:

1. *Thou shalt not read upward; and*
2. *Thou shalt not write downward.*

The adverbs "upward" and "downward" refer to levels of security.

It is implied that every actor in a security transaction possesses an attribute called "clearance" which is quantified as a level of security. Within a computer system the actor may be an interactive user or an executing program.

Likewise, it is implied that every object in a security transaction possesses an attribute called "classification" which is similarly quantified as a level of security. Within a computer system, an object is a data item whose length may vary from a single bit to a countably infinite number of bits.

*A non-discretionary security model postulates several discrete levels. These commonly range (in the Canadian context) from un-*classified *or* level-0 *to top secret or* level-4.

In the non-discretionary data-flow security model, an executing program or process must possess a secret clearance to seize an object possessing a secret classification. Moreover, any other object that is functionally related to a secret object must, thereafter, attract a secret classification.

This is known as "high-water-mark" classification of objects. It is viewed, especially in the private sector of the economy, as causing expensive "overclassification" of files and outputs.

It is proposed that a cost-effective solution to the problem of overclassification could rest upon the principle of "work-factor" security.

The work-factor is a measure of the work that an opponent would have to do to recover protected information from a less highly classified object that is functionally related to a more highly classified object. If that work is more expensive to the opponent than the value to him of the protected information, then cost/ effective work-factor security will have been achieved. The opponent will be deterred if he is the rational economic man encountered in the private sector of the economy.

The first step is to represent the data-processing system by a data-flow-diagram in accordance with standards adopted by the Canadian Department of National Defence.

There is an underlying assumption that only trusted processes are used. That is, that every program in the system does what it is intended to do: no more and no less.

The second step is to resolve the processing nodes or "bubbles" of the data-flow-diagram in terms of relational algebra primitives. It is convenient to observe the restriction that the inputs or outputs of the functions be either monadic (m) *or dyadic* (d).

The third step is to prepare a PROLOG data base of all inputs and outputs of all functions.

The fourth step is to prepare a PROLOG data base of all functions and all outputs.

The fifth step is to prepare a PROLOG data base of work-factor weights. One entry is made for each unique function / weight combination.

Work-factor weights are awarded empirically. The higher the weight, the more the function under consideration contributes to obscuring the protected data in a specific output. Furthermore, the weight values must be decided with respect to a specific data item.

Consider a case in which a highly classified data item is called

customer-credit-file

Within the data-processing system there is a PROLOG function that strips the cash-sale / credit-sale flag from every record in a file of good-sales-orders (that is, either cash sales or sales to credit-worthy customers). The results of this operation are : (1) a file called untagged-sales-orders, and (2) another appearance of the original file, called good-sales-order-2.

The output data base entries would be :

```
next(good-sales-order-1, good-sales-order-2).
next(good-sales-order-1, untagged-sales-order).
```

The identifier md *below means that the operation in question produces either a monadic* (m) *or dyadic* (d) *output.*

The function data base entries would be :

```
opr(untagged-sales-order, project2,md).
opr(good-sales-order, project1,md).
```

The weight data base entries would be :

```
wght(project1,md,1).
wght(project2,md,3).
```

The rationale for assigning different weights is as follows : When good-sales-orders are flagged as being cash or credit, an opponent can infer that the customers flagged as credit sales had favourable entries in the highly classified customer-credit-file. Therefore, protected information would be compromised if that opponent could view good-sales-orders-1. For this reason a work-factor weight of "one" is assigned signifying no protection. On the other hand, an opponent could obtain customer credit data from untagged sales only by exhaustively making spurious purchases on credit using the names of different customers, and recording which sales were considered to be good ; thereby inferring that the other customers had unfavourable credit records. Such an attack, which corresponds to a chosen plaintext attack in cryptanalysis, is expensive and time-consuming. Only a resourceful, dedicated and implacable opponent would undertake it. For most adversaries its cost would greatly exceed the value of the information derived. For this reason, a work-factor of "three" has been assigned.

The choice of work-factor value is judgmental. The value can range from "one", signifying no protection, to "ten" which might be assigned to the encrypted output of an encrypting CALCULATE function.

The PROLOG program is recursive. Its purpose is to exhaustively enumerate all descendents of a sensitive file in a complex data-processing system ; this is done so that appropriate measures can be taken to safeguard or disguise these descendents, and thereby preserve the security of the sensitive file.

```
descendent(X,Y,W,D,O) : –
        next(X,Y), opr(Y,O,T),
        wght(O,T,W,), depth(D).

descendent(X,Z,W,D,P) : –
        next(X,Y), opr(Y,O,T),
        wght(O,T,M,), descendent(Y,Z,L,D,P)
        W is M*L.
```

In the PROLOG program, the weight assigned to a data item is the lowest product of weights assigned over the paths from the protected data item to it. The effect of this rule is exemplified by the following case. Further along in the example, there is a VALIDATE function which checks part numbers and quantities ordered against part numbers and inventory balances on hand. The output consists of fulfillable-sales-orders and an insufficient-stock-report. The fulfillable-sales-orders output is assigned a work-factor weight of "two". The overall work factor of this output is "six".

It is reasoned that the fact that : (1) some orders are not filled because the customers want to buy on credit and have un-favourable credit records, and (2) some orders, whether credit or cash, are not filled because the supplier is out of stock would confound an opponent even if he were to launch a controlled plain-text attack. The two work factors combine multiplicatively to make life more difficult for an opponent.

The PROLOG question is :

```
?-descendent(customer-credit-file,X,Weight,Depth,Opr).
```

The first term is the name of the highly classified data item, the second term evokes all descendents of that file, the third term gives the work-factor protection of that item, the fourth term indicates the depth of the tree or the "distance" in a data-processing sense between the highly classified data item and the item in question, and the fifth term identifies the function producing the data item.

The answers after the PROJECT function removes the credit / cash flag are :

```
Depth = 4
Opr = Project2,
Weight = 3,
X = untagged-sales-order ;

Depth = 4
Opr = Project1,
Weight = 1,
X = good-sales-order-2 ;
```

The file good-sales-order-2 *is scratched and therefore represents no security exposure.*

The answers after the VALIDATE function has been invoked to check inventory status are:

```
Depth = 8
Opr = validate2,
Weight = 6,
X = insufficient-stock-report ;

Depth = 8
Opr = validate2,
Weight = 6,
X = fulfillable-sales-order-1 ;
```

Every data item that is functionally related to the protected item is identified by the program. Moreover, each of these items is assigned a weight that is a measure of the work an opponent would have to do to obtain knowledge of the contents of the protected item given that he has gained access to the item in question.

Analyses using PROLOG have also been carried out on a payroll preparation system where the highly classified data items were employee salaries, and on a CAD/CAM (computer-aided-design/ /computer-aided-manufacturing) system in which the highly classified data item was the description of a proprietary machining operation.

It is believed that PROLOG analysis of complex data-processing systems can assist greatly in applying cost-effective security measures that will protect "trade secrets" and personal data such as salaries and credit ratings in the private sector. It can also disclose potential security exposures in the public sector, where in sensitive systems, the weights might be all set to "one"; or "ten" where cryptographic protection is invoked.

WARREN'S DOCTRINE ON THE SLASH

M. H. van Emden

Department of Computing
Imperial College *, UK

It is often stated that there are two different uses of the slash: an innocent one conveying only control information and a harmful use affecting the meaning of the program. Warren has repeatedly explained in conversation that any use of the slash is only control information and that it is not useful to say that it "affects meaning", whatever that means. It seems that Warren's doctrine is sufficiently important and neglected to warrant this brief note.

The Doctrine states that the predicate "/" has a nullary relation as meaning, as indeed its syntax suggests. There are two nullary relations, FALSE and TRUE. The Doctrine states that the meaning of "/" is TRUE. In Prolog there is a side effect of proving the occurrence of a goal "/" in the program and that is to remove a part of the search tree. Which part is removed is explained by Prolog manuals and need not concern us here. The role of "/" is analogous to that of output predicates: succeed always and have a side effect.

Thus according to the Doctrine, a Prolog program with slashes has meaning according the classical, Tarskian semantics of logic and the meaning remains the same when any or all slashes are removed. Now, in what sense is there, according to the Doctrine, a problem?

Logic programming is useful because it supports correctness--oriented programming to a greater extent than other languages. That is, each clause of a logic program must be true of the relation to be computed. When this is the case, correctness of the interpreter guarantees that no falsehood can result from an activation of a Prolog program. When a false clause does occur in a program, falsehood can result. But these falsehoods can be prevented from appearing if certain parts of the search tree are removed. And this can be done with a slash. See the classical example

```
max(X,Y,Y) <— X<Y & /.
max(X,Y,X).
```

The second clause is not true of the relation where the third argument is the maximum of the first two. The slash prevents any consequences of its falsity from appearing.

It is not the purpose of this note to discuss proper use of the slash. See for example Kowalski's "Prolog as a Language for Logic Programming" (Research Report DOC 81/26, Imperial College). The only purpose of this note is to explain Warren's doctrine:

"/" denotes TRUE

and to point out that is always contains control information only, which can be used to prevent the appearance of erroneous consequences of false clauses.

Thus the Doctrine indeed distinguishes two kinds of use of slash. In one case the clauses are true of the relation to be computed and in the other case they are not. The latter use is incompatible with correctness-oriented programming and may therefore be considered harmful.

After disposing of the main business of this note it is useful to speculate on why harmful uses of slash are so common. In the examples that I can think of at the moment, the effect of the false clauses is always to define a relation containing *the intended one. Because the slash removes part of the search tree, it causes the interpreter to compute a* subset *of the meaning (minimal model) of the program. Apparently, the limitations of pure Prolog can often be circumvented by writing a false program defining a superset of the intended relations and then, by craftily positioned slashes, to prune those parts from the search tree which contain false answers.*

The following heuristic is sometimes helpful in solving probability problems: "If it turns out to be difficult to determine the probability that something happens, then try to find the probability that it does not". Prolog programmers seem to apply a similar trick: to get the right set of answers, it may be easier to define a relation that is larger than the desired one and to lop off the undesired part.

P. S. *"!" or "/"? An advantage of the latter is that the usual name of the symbol suggests its intended use.*

* On leave from the Dept. of Computer Science, University of Waterloo.

TEACHING COMPANIES AND SNOWBIRD COMPANIES

There are two kinds of hospitals: you have the ordinary kind and then there are teaching hospitals. The latter are associated with university medical schools. The faculty do the pratical aspects of their teaching in that kind of hospital as well as their research.

M. Lehman, head of the Department of Computing in London's Imperial College, has [Comm. ACM October 1981, pp. 718-719] proposed the concept of a «teaching company». The example he has in mind is a software house, which has the same relationship to an ordinary software house as a teaching hospital has to an ordinary hospital. There is a letter from Lehman in the October 1981 issue of the Comm. ACM explaining his plans in this regard.

Another interesting news item reports on new companies formed to exploit pratical applications of genetic engineering. One of these is called Cetus and has its head office in Berkeley, California. Winston J. Brill, one of the key people they wanted for their new labis a faculty member in the University of Wisconsin. The interesting phenomenon is that prof. Brill is now working for Cetus without having been bought away from his University. This company was apparently flexible enough to arrange for a small Cetus lab to be established just off the University of Wisconsin's campus. Now they not only have prof. Brill working for them, but also some of his students [Sci. Am. Sept. 1981, Feb. 1982].

The manpower problems in computer science are similar to those in molecular genetics and have been publicized in the reports of a meeting of Computer Science Department chairmen in Snowbird, Utah in the summer of 1980 (see Comm. ACM June 1981). The situation seems to be that industry and government think they need each year about three times as many computer science PhD's as are produced by universities. It does not matter whether they are indeed justified in claiming this need. They feel the need and they act accordingly, creating the well-known problems in universities. The above two examples suggest a definite strategy on the part of individual computer science faculty nembers.

They should go out to government and industry and find out what it is specifically in their area of expertise that causes the perceived need for PhD's. It may well be that certain planned research or development projects emerge. And it may be possible to persuade the planners to have some of this work done on a contract basis in a small company (I propose to call it a «Snowbird Company») which is off-campus with respect to a computer science department rich in human resources. The recent development of powerful micro computers and improved data communication facilities makes such a set-up feasible in many cases.

Snowbird companies spread benefits all around: industry gets its work done, needs fewer PhD's, and university departments become more attractive to good PhD's in computer science. Students benefit by employment opportunities in a more innovative environment than that of the typical coop job. They benefit also by having a larger proportion of practical, software-oriented faculty members.

INTRODUCING SNOWBIRD COMPUTING CONSULTANTS, Ltd.

There is a severe shortage of computer science PhD's: authoritative estimates put the supply at less than half of demand. In Artificial Intelligence the situation seems to be at least as extreme. This imbalance creates well-known problems both in industry and in universities. Snowbird Computing Consultants Ltd, which has been founded to organize the consulting activities of a small group of university faculty nembers and their students, is an example of a type of company which can help solve these problems. Each in its own field, such companies determine which specific planned or ongoing projects are in need of unavailable PhD's. They offer contributions to such projects on a consultancy basis. Clients benefit by getting their job done. Universities benefit by having fewer faculty members bought away by industry and by having faculty and students gain practical experience at an advanced technical level.

Snowbird is based in Waterloo, Ontario, Canada. It consults in Artificial Intelligence and Logic Programming. Entry in these fields is especially timely.

Artificial Intelligence arrives

For many years, Artificial Intelligence has remained a rather esoteric branch of Computer Science, more readily associated with robots in space than with mundane concerns like making computer applications more accessible to nonspecialist end-users. Yet it is precisely here that Artificial Intelligence is most promising.

In the near future all professionals, most without any computing experience, have access to computers. Current interfaces do not allow this potential to be realized because supporting computer specialists are not available. Data Bases, for example, may have grown Very Large, but are not informative or helpful to the casual user; extremely sophisticated program libraries exist for statistics, numerical computation, structural engineering, etc., but most scientists and engineers, for whose use they are intended, cannot benefit without an expert as intermediary. Several contributions from Artificial Intelligence have recently been shown to be of immediate practical use in bridging the gap between the non-specialist user and computer applications. These contributions include non-procedural languages, expert systems, natural language interfaces, and knowledge representation techniques.

Very few applications of Artificial Intelligence are in actual use. One important reason has been its high demands in terms of computing power. In recent years hardware has become much less of a bottleneck. Yet, because Artificial Intelligence is almost exclusively practiced on very demanding Lisp systems, a problem remains.

Logic Programming — an idea whose time has come

Logic programming is the use of logic to

* express information in a computer
* to present problems to a computer
* to use logical inference to solve these problems

An important part of the aims of logic programming is realized by the Prolog language. Because of its bases in logic, Prolog resembles human knowledge representations more closely than Lisp. This makes Prolog a very effective language for implementing the techniques of Artificial Intelligence. Efficient implementations of Prolog are beginning to become available. Notable, for example, is the microProlog system from Logic Programming Associates of London, England, which runs on a Z80 microprocessor and can do some useful problems within 64K bytes of storage. *The full benefits of Prolog can be realized on the new generation of 16-bit microcomputers.*

community news & events

The role of Snowbird Computing Consultants, Ltd.

Logic programming has been developed by a number of independent workers maintaining close contact while associated with universities and laboratories dispersed throughout Europe and, to a lesser extent, North America. Although the University of Waterloo has played but a modest role in this development, it has, over the past 6 years, built up a valuable store of expertise and contacts in this area. While research into the theoretical aspects of logic programming continues to be based at the University of Waterloo, Snowbird is available to Waterloo faculty and students as a vehicle for application-oriented work.

Although not formally associated with Snowbird, many prominent workers in logic programming are in principle available as consultants.

Snowbird offers consultancy in

* intelligent databases
* natural language interfaces
* expert systems

Snowbird also conducts seminars and individual training in the practice of logic programming, in Prolog implementation, and in application of logic programming in areas such as specification techniques, program verification, and semantics of programming languages.

The projects we contemplate fall into two distinct categories. The first consists of applications using existing Prolog implementations. In the other, the application is sufficiently demanding to require at least adaptation of an existing Prolog implementation, for example to incorporate sizable data bases, to perform coroutining, or to interface efficiently with low-level software. Experience shows that, with a suitable operating system, such as Unix, the required adaptation can be completed with modest effort.

For more information, contact Randy Goebel or Maarten van Emden at Snowbird, 229 Dick Street, Waterloo, Ontario NZL 1N3, Canada.

APPIA

The «Associação Portuguesa Para a Inteligência Artificial» (APPIA) has been created. Send any mail to the chairman, Luís Moniz Pereira.

ANNOUNCEMENT

Michel van Caneghem and David Warren are to co-edit a book of papers on logic programming. The book will be published by Ablex Publishing Company as a volume in the series entitled «Advances in Artificial Intelligence» under the general editorship of Jerry Hobbs. The editors are seeking high quality papers on all aspects of logic programming, especially applications. Please send copies of manuscripts to at least one and preferably both of the co-editors:

David Warren
Artificial Intelligence Center
SRI International
333 Ravenswood Avenue
Menlo Park
California 94025, USA

Michel van Caneghem
Groupe d'Intelligence Artificielle
Faculté des Sciences de Luminy, case 901
13288 Marseille cedex 9
France

If a manuscript is identical to the version of a paper which appeared in the Proceedings of the First International Logic Programming Conference at Marseille, the author need only notify the co-editors that he wishes it to be considered for the book.

NEWS FROM SRI INTERNATIONAL

Fernando Pereira joined the AI Center on September 13 as a Computer Scientist to work on natural language and logic programming.

Bill Zaumen, of the Telecommunication Sciences Center, has developed a Prolog program to analyse VLSI circuits and check them for «syntatic» errors.

CHANGE OF ADDRESS

Ehud Shapiro, formerly at Yale, has completed his PhD. and is now at Dept. of Applied Mathematics, Weizmann Institute, Rehovot 7600, Israel.

ECCAI GOT STARTED

After two and a half years of initiatives and exchange of views, a number of representatives of European Artificial Intelligence societies has now agreed to start the European Co-ordinating Committee on Artificial Intelligence. In a constitutional meeting held during ECAI-82 the statutes of ECCAI, presented below, have unanimously been approved and the officers listed at the bottom of this page have been elected.

Since so much work lies now ahead of us, I will not spend the time with talking about the process which eventually led to this action. I would like, however, to point out the active and fruitful role which was played in the preparatory phase by Dennis de Champeaux, Amsterdam/New Orleans, who was appointed in 1980 for setting up ECCAI. From the material he handed over to me, I can see how much energy he invested in order to achieve this success.

I would like to encourage further European AI communities to join ECCAI, in order to provide it with a broader basis and thus with more strength. This might require that such communities first find together in an organized group or society.

Besides simply getting started the first concrete actions of ECCAI have been mainly the following ones.

The next European AI conference, ECAI-84, is planned to be held in Pisa (and ECAI-86 in the U.K.).

A subcommittee has been set up for EEC matters such as the promotion of a European network which could be used by all AI groups. Its members are L. Aiello, Rome, J. P. Jouannaud, Nancy, P. Raulefs, Kaiserslautern (chairman), and Y. Wilks, Colchester.

The Commission of the European Communities in Brussels has various programmes for supporting data processing and advanced information technology in general which explicitly includes artificial intelligence.

In the framework of its «pluriannual programme», for instance, support is offered for more cooperation in AI within Europe. The CALL FOR PROPOSALS is presented below.

A second instance is an industrial programme running under the keyword ESPRIT which may be seen as a European response to Japan's 5G project.

It is designed for a period of ten years, with starter projects planned to be initiated next year. Many of its goals are typically of an AI nature. Therefore we feel that European AI research should reasonably be participated.

P. Raulefs and I are working towards such goals in the respective Scientific Council at the Commission.

Finally, we are planning to gather information about AI activities in Europe to be made available in a pamphlet and would appreciate any relevant input.

With these initiatives we hope to convince the AI community that ECCAI is not just another burocratic body, rather might play a fruitful role in promoting AI research within Europe. We hope for the strong support from the side of the community in this respect.

W. Bibel — chairman

EUROPEAN CO-ORDINATING COMMITTEE FOR ARTIFICIAL INTELLIGENCE

STATUTES

Aim. To promote the study and application of Artificial Intelligence in Europe.

Membership. European Artificial Intelligence Societies, namely:

AISB	Society for Artificial Intelligence and Simulation of Behaviour
GI	Gesellschaft für Informatik
AFCET	Association Française de Cybernétique, Economie et Technologie
AICA	Associazione Italiana per il Calcolo Automatico
ARC	Association pour la Recherche Cognitive
NVKI	Nederlandse Vereiniging vor Kunst matige Intelligentie
OGAI	Österreichische Gesellschaft für Artificial Intelligence.
APPIA	Associação Portuguesa Para a Inteligência Artificial (proposed).

New societies of more than 25 members are to be admitted by a 2/3 vote of the existing membership.

Societies of less than 25 members may be admitted as non-voting participants by a 2/3 vote of the existing membership.

Representation. Each society shall be represented by two persons nominated by the society.

Officers. The committee shall have a chairman, and a secretary elected for two-year terms by the members of the committee.

Procedures. The chairman shall convene committee meetings at least annually. The secretary shall run elections. All initiatives including changes to the constitution of the committee must be supported by a majority of 2/3 of the attendance at committee meetings. There must be a minimum of one month's notice for meetings. Proposals must be circulated with the notice of the meeting. Voting rights may be delegated with the notice of the meeting. Voting rights may be delegated or exercised postally.

Activities

i) To sponsor a biennial conference to be organised by one or more of the participating societies.
ii) To create subcommittees with special terms of reference including a subcom mittee concerned with EEC issues of common market based societies.
iii) To promote the construction of a European network.
iv) To carry out other activities as appropriate.

CALL FOR PROPOSALS FOR COLLABORATIVE RESEARCH WORK IN THE FIELD OF ARTIFICIAL INTELLIGENCE AND PATTERN RECOCNITION

Objective

The Commission of the European Communities, acting in the framework of the pluriannual programme of the Community in data processing has decided to grant financial support to cooperative research work in the field of Artificial Intelligence and Pattern Recognition.

The assistance is not intended to support the cost of the basic research activity carried out by a research team, but rather to suport the additional costs of collaboration between teams in different Community countries: travel, subsistence when working abroad, joint workshops, usage of network services and so on. In exceptional cases a more substantial contribution could also be given to projects of particular interest.

Topics

Among the various fields of prospective interest preference will be given initially to those proposals which cover one of the following:

Domains:
— Image processing and understanting
— Speech processing and understanding

Techniques:
— New algorithms
— Parallel processing
— Expert systems

Applications:
— Robotics
— Office automation
— Medicine
— Remote sensing.

Submission of proposals

Research teams or individuals working in these areas are invited to present proposals for cooperative international research.

Proposals for financial assistance should state

— the objectives and the programme of work for which support is sought and its relationship to the state of the art in the area.
— the resources (financial and staff) which will be made available for the work with some indication on the way they are to be organized.
— the ways and means of the planned cooperation and the expected benefits to be gained through international cooperation.
— the possible links with other work.
— the aid requested and its justification.
— the timetable for the work and for the payment of aid.
— any other information considered relevant for the appraisal of the proposal.

Should the detailed definition of the proposal require expenditure on travel that cannot be met by the proposer from national resources, a preliminary proposal may be made to cover a contribution to these costs.

The proposals should be addressed before 1 January 1983 to:

The Commission of the European Communities
Directorate-General III
Attention Mr J. Desfosses
200, rue de la loi
B — 1049 Brussels, Belgium

In return for its aid, the Commission may require progress reports and a commitment to publish the results.

The selection of the proposals will be made with the assistance of the CREST Subcommittee on «Informatics and Information Technologies».

The Commission reserves its right to refuse any proposal without any justification.

new books

Beginning micro-PROLOG

RICHARD ENNALS
Department of Computing, Imperial College of Science and Technology
University of London

This book approaches PROLOG, the new microcomputer language (PROgramming in LOGic), from the point of the human user. Written for the non-specialist as well as the computer professional teacher, or student user, the book is based on material developed from a project, Logic as a Computer Language for Children, run by innovator of the language, Dr. Robert Kowalski. The project enjoyed the full support of the Science and Engineering Research Council of Great Britain, as well as the Nuffield Foundation.

The text is designed for use with, or without, a microcomputer, and contains exercises and teaching material for class use. Each new feature of the language introduced is ilustrated in programs written by schoolchildren from the original Pilot School. The project is described in the context of recent developments in both educational computing and logic programming, and the language is introduced in the same clear manner as in the classroom. It raises the issue of the application of logic across the curriculum, with explorations into the areas of Information Technology, Mathematics, History, French and Science teaching. The final section looks ahead to future opportunities for the layman provided by logic programming in areas such as expert systems and large data base applications, and in the new «fifth generation» of computers that are now being planned.

The Author

Richard Ennals is a Research Fellow in the Department of Computing at Imperial College of Science and Technology, employed on the project «Logic as a Computer Language for Children», and working in the Logic Programming Group led by Dr. Kowalski. He was an English scholar at Kings College Cambridge; English-speaking Union Exchange Scholar at Philips Academy, Andover, Massachusetts; read Philosophy and History at Cambridge; trained as a History Teacher at the Institute of Education, University of London; was Head of History at Sweyne School, Rayleigh, Essex; has lectured widely in schools, colleges and universities in Britain, America, Canada and Nigeria; and has published a number of articles on history, simulations and applications of PROLOG.

The Language

Micro-PROLOG is currently available on microcomputers using the Z80 microprocessor and the CP/M Operating System, such as Research Machines 380Z, North Star Horizon and Advantage, Zenith, Transam, Apple (with Z80 card and upper and lower case characters). It is now being implemented on the Sinclair Spectrum, and will soon be available on the BBC Micro, Commodore PET, and Apple II.

Order from

Department MP
ELLIS HORWOOD LIMITED — PUBLISHERS
Market Cross House
Cooper Street
Chichester
West Sussex PO19 1EB
England

ISBN 0-85312-517-1 *approx. 200 pages (paperback)* *£7.50*

abstracts

A View of the Fifth Generation and its Impact

David H. D. Warren

Artificial Intelligence Center,
SRI International
Menlo Park, CA 94025, USA

In October 1981, Japan announced a national project to develop highly innovative computer systems for the 1990s, with the title «Fifth Generation Computer Systems». This paper is a personal view of that project, its significance, and reactions to it.

Interfacing Predicate Logic Languages and Relational Databases

Upen S. Chakravarthy, Jack Minker and *Duc Tran*

University of Maryland
College Park, Maryland 20742, USA

Predicate logic has been shown to be an effective language for expressing knowledge and for problem solving. It has also been recognized that predicate logic and relational databases (RDB) are closely related. A predicate logic language such as PROLOG can be used to perform queries on relational databases.

Current implementations of PROLOG assume that the RDB resides in primary memory. However, when the RDB resides in peripheral devices, the utility of PROLOG diminishes.

We describe several ways in which PROLOG-like systems can be modified to operate effectively with large RDBs stored on peripheral devices. Two approaches to interfacing a PROLOG-like system with large databases are the compiled approach described independently by Chang, by Kellogg, Klahr and Travis, and by Reiter, and the interpreter approach as discussed by Minker. We show how PROLOG can be adapted for the so-called compiled approach. With respect to the interpreter approach we discuss a method

proposed by Bruynooghe which modifies PROLOG in a simple manner. An approach using tables at search nodes of a proof tree is then described. The table approach requires a more extensive modification of PROLOG than the other approaches. We discuss the concept of table-sharing, which is similar to structure sharing in many respects.

On a Semantic Representation of Natural Language Sentences

Jean François Pique

Lab. de Biophysique,
Faculté de Médecine Timone,
Université d'Aix-Marseille II, France

In this paper we consider problems in the hierarchy of sentence parts. Some solutions are given, representing the meaning in a three valued logic which we briefly describe.

Coloring Maps and the Kowalski Doctrine

John McCarthy

Dept. of Computer Science
Stanford University, California, USA

It is attractive to regard an algorithm as composed of the logic determining what the results are and the control determining how the result is obtained. Logic programmers like to regard programming as controlled deduction, and there have been several proposais for controlling the deduction expressed by a Prolog program and not always using Prolog's normal backtracking algorithm. The present note discusses a map coloring program proposed by Pereira and Porto and two coloring algorithms that can be regarded as control applied to its logic. However, the control mechanisms required go far beyond those that have been contemplated in the Prolog literature.

Relational Data Bases «à la carte»

Luís Moniz Pereira
Miguel Filgueiras

Departamento de Informática
Universidade Nova de Lisboa
Quinta da Torre
2825 Monte da Caparica, Portugal

In some cases access to a relational data base by means of a menu may be the best solution compatible with the requeriments on the access proper as well as with the time and money restrictions on the making of an access program.

We have developed a general program, in Prolog, that using information gathered from the user generates a menu based consultation program tailored to suit the user data base and acess requirements. Some general ideas on relational data base access were used concerning implicit fields, combining of fields, special access predicates, references to texts, finding names from partial descriptions, etc.

We claim the high usefulness of this program. To those who have data to store and retrieve, their work is only to plan a relational data base. The access program is instantly made.

On the Implementation of Control in Logic Programming Languages

Miguel Filgueiras

Departamento de Informática
Universidade Nova de Lisboa
Quinta da Torre
2825 Monte da Caparica, Portugal

First a description is given of some experiments on a (simulated) low-level implementation for a particular case of control. Then, a comparison concerning complexity and efficiency is made between this approach and a high-level implementation (in Prolog) for the same case. Conclusions on the more effective implementation level of logic programming languages introducing sequential parallel and co-routined execution are then presented as a generalization of the results for the particular case studied.

NON-VON: a Parallel Machine Architecture for Knowledge-Based Information Processing

David E. Shaw

Department of Computer Science
Columbia University
New York, NY 10027, USA

NON-VON is a highly parallel machine designed to support the efficient implementation of very large scale knowledge-based systems. The utility of such a machine has been demonstrated analytically, and through implementation of a working knowledge--based information retrieval system in which the NON-VON machine instructions were emulated in software. In cooperation with the Stanford Computer Science Department, we have begun to implement the most important components of the machine as custom VLSI circuits.

The NON-VON Database Machine: A Brief Overview

David E. Shaw
Salvatore J. Stolfo
Hussein Ibrahim
Bruce Hillyer

Department of Computer Science
Columbia University,
New York, NY 10027, USA

Gio Wiederhold
J. A. Andrews

Department of Computer Science
Stanford University, California, USA

The NON-VON machine (portions of which are presently under construction in the Department of Computer Science at Columbia, in cooperation with the Knowledge Base Management Systems Project at Stanford) was designed to apply computational parallelism on a rather massive scale to a large share of the information processing functions now performed by digital computers.

The NON-VON architecture comprises a tree-structured Primary Processing Subsystem (PPS), which we are implementing using custom nMOS VLSI chips, and a Secondary Processing Subsystem (SPS) incorporating modified, highly intelligent disk drives. NON-VON should permit particularly dramatic performance improvenents in very large scale data manipulation tasks, including relational database operations and external sorting. This paper includes a brief overview of the NON-VON machine and a more detailed discussion of the structure and function of the PPS unit and its constituent processing subsystems.

abstracts

On the long-term implications of database machine research

David E. Shaw

Department of Computer Science
Columbia University
New York, NY 10027, USA

It is argued that the long-term import of current research on specialized hardware for database management may extend far beyond what is now considered the immediate province of database management systems. In particular, the potential support which «database machines» may provide for very high level mechanisms for the description of information processing tasks is considered and exemplified with an operational system for knowledge-based information retrieval.

A Partial Evaluator of Lisp Programs Written in Prolog

Kenneth M. Kahn

Department of Computing Science,
Uppsala University
750 02 Uppsala, Sweden

A new experimental technique for automatically generating compilers from interpreters is applied to a Prolog interpreter. The Prolog interpreter called LM-Prolog was implemented in Lisp, with the dual purpose of supporting a partial evaluator for Lisp programs called Partial and that the implementation itself be a suitable input to Partial. A partial evaluator is a program which takes in programs and generates more efficient, less general, versions of them. Partial is being applied to LM-Prolog to produce specialized versions of Prolog for a given database of assertions or programs. If successful, an efficient compiler/interpreter Prolog system will be produced based upon a small simple interpreter. Maintenance and development of LM-Prolog is confined completely to the interpreter and the compiler will automatically be kept compatible. Additionally, compilers for any interpreters built upon LM-Prolog can be automatically generated.

Concurrent Execution of Logic

Kenneth A. Bowen

School of Computer & Information Science
Syracuse University, Syracuse, NY 13210 USA

A preliminary study of a multi-processor interpreter for PROLOG is presented. The processors involved share no common memory and communicate by message passing. The interpreter provides for parallelism at both AND- and OR- nodes in PROLOG search trees. The development is top-down, beginning from search in an abstract tree. Algorithms are expressed in modified MODULA.

Prolog Interpreter Based on Concurrent Progamming

Koichi Furukawa

ICOT — Institute for New Generation
Computer Technology
Tokyo 108, Japan

Katsumi Nitta, Yuji Matsumoto

Electrotechnical Laboratory
Ibaraki 305, Japan

This paper presents a coroutine-based algorithm for interpreting Prolog programs. Two kinds of processes, named AND-process and OR-process, are introduced. It turns out that the complexity of an interpreter decreases by dividing tasks properly into each of them. We informally prove the correctness of our interpreter. The control part and unification part are separately proven. Finally we present a possibility to incorporate highly parallel execution of Prolog programs.

A Lisp-Machine to Implement Prolog

C. Percebois
J. P. Sansonnet

Laboratoire «Langages et Systèmes Informatiques»
Université Paul Sabatier
116 route de Narbonne
31062 Toulouse Cedex, France

If throughout these past few years, new architectures have been developed in order to reduce the semantic gap separating the software from the hardware, too many applications yet require a difficult implementing effort.

Through a direct execution model whose internal form derived from LISP, we have been able to bridge this gap thanks to an architecture embodying directly the concepts of a high level language specialized in tree-structured processing.

We present here an outline of a PROLOG implementation for this architecture. Then we give evaluation measures dealing with size and speed of our future interpreter.

An Interpreting Algorithm for Prolog Programs

M. H. van Emden

Department of Computer Science,
University of Waterloo, Canada

We describe in pseudocode the main routine of a Prolog interpreter. The algorithm is developed in several steps. Initially a search algorithm for trees is described. After a review of the Prolog theorem prover, the tree-search algorithm is applied to the search space of the theorem prover. Several inefficiencies of the result are then eliminated by the introduction of proof trees and structure sharing.

Prolog/KR — Language Features

Hideyuki Nakashima

Information Engineering Course
University of Tokyo, Japan

Language features of Prolog/KR are described. The language is mainly based on Prolog but has been extended to improve Prolog's capability of expressing controls. Although most implementations of Prolog support a cut operator to suppress global backtracking, it is too primitive to construct a large system. Prolog/KR provides, instead of the cut operator, plenty of higher order control predicates to make control structures manifest. The control structure also includes parallel computation and co-routines. Besides the control structures, Prolog/KR has the following features:

1. The pattern matcher (unifier) has been extended.
2. The system apprehends multiple worlds.
3. Substantial debugging facilities are supplied.

abstracts

Logic Programming: A Parallel Approach

Norbert Eisinger, Simon Kasif, Jack Minker

Computer Science Dept., Univ. of Maryland, College Park, MD 20742, USA

Predicate logic programming provides a convenient mechanism for problem solving in the context of a multiprocessor system. The writer of a logic program need not be concerned about the specific configuration of the system, and advantage can be taken of inherent parallelism that exists in the system. Multiple problem solving machines may be working simultaneously on the same problem, sending requests to the set of machines containing procedure statements to match procedure heads and return responses. The system, described in general is being designed for a specific multiprocessor system, ZMOB. The system consists of a ring of 256 Z80A microprocessors interconnected on a high speed bus together with the VAX-11 computer.

Knowledge Acquisition in Prolog

Alain Grumbach

Laboratoires de Marcoussis (CRCGE)
Route de Nozay 91460 Marcoussis, France

The problem we are dealing with is the problem of updating data or knowledge bases.

We would like to provide the user with tools such that updating could be done in the same natural declarative manner as the queries. 3 kinds of logic requests would be possible:

query:	«who teaches course2 ?»
assertion:	«it is true that course2 is lec tured by linda.»
negation:	«it is false that course3 holds on monday».

The user will not have to know anything about the internal informations.

PROLOG appears to be very well suited to this kind of problem, (in comparison with other powerful languages as LISP or PASCAL), owing to the fact that facts and rules are represented in the same way, and to the unification technique.

We describe the corresponding feature on an example dealing with semantic parsing of natural language sentences, and then apply it to the updating of relational data bases and knowledge based systems.

Teaching Logic as a Computer Language in Schools

Richard Ennals

Department of Computing
Imperial College of Science & Technology,
London SW7 2BZ, UK

The experience described in this paper has been gained on the project «Logic as a Computer Language for Children» which is supported by the Science and Engineering Research Council and the Nuffield Foundation. It is based at Imperial College, and has involved regular classroom teaching at Park House Middle School, Wimbledon since September 1980. An account will be given of the progress of the initial pilot class, the revisions that have been made for subsequent classes, the experimental teaching of mathematics and history using PROLOG, the development of a «logic across the curriculum» approach to the teaching of mathematics, history, science and French, the development of an Information Technology course based on PROLOG, and a course using different PROLOG implementations. Issues for discussion will include modes of representation, levels of logical reasoning, the practicalities of programming, the importance or otherwise of efficiency and specification, and suggestions for future research.

Graphs as Data in PROLOG Programs

Jan Šebelík

Institute for Application of Computing Technique in Control, Prague 1,
Revoluční 24, Czechoslovakia

Petr Štěpánek

Department of Cybernetics and Operational Research, Charles University, Prague 1,
Malostranské náměstí 25, Czechoslovakia

Three types of representations of binary graphs in PROLOG programs are described, namely, graphs are represented as specific terms, by listing all edges and by collection of complete subgraphs. It is shown that there are some conditions on the specific representations of graphs, which make possible to construct higher level logic programs for solving some classical graph theoretic problems, which are independent of the representation. The communication with the specific data representation is realized by two so called fundamental predicates. Representation dependent conversion programs are described as well.

A Prolog Simulation of Migration Decision Making in a Less Developed Country

John W. Roach

Department of Computer Science

Theodore D. Fuller

Department of Sociology
Virginia Polytechnic Institute
and State University, Virginia, USA

The logic programing language PROLOG is used to create a cognitive model for migration decision making among Thai villagers. The simulation includes representations of social mores as well as of personal values written in PROLOG rules. Several prototypical Thai villagers are assigned characteristics (derived from real data) and run through the simulation to determine whether they will decide to migrate to the city. The relevance of working models, programs embodying a theory, to the social sciences is discussed. We conclude that working models have some advantages over more traditional statistical models: with social rules implemented in a rule-based logic language, we can assess the adequacy of the social theory by observing the behavior of prototypes. Possible future expansions to the program are suggested.

Module Development Based on Program Transformation and Automatic Generation of the Input-Output Relation

Douglas R. Skuce

Department of Gomputer Science
University of Ottawa
Ottawa, Ontario, Canada

A practical methodology, termed «generator based programming», for developing relia-

ble software modules is proposed, which synthesizes ideas from logic programming, detaflow programming, automatic tresting, and program transformation. In brief: a module M is designed as a series of submodules; each accepts as input a stream of input tokens and emits another stream as inputs to the next submodule. The first submodule is initially designed as a generator of its input-output relation, i.e all pairs of its input and output streams. A list of these can be quickly inspected to reveal errors in the relation, to give the writer confidence that the intended relation has indeed been obtained. We stress the importance of this «intentional feedback». The generator of subsequent modules, and lists of input-output pairs may thus be automatically generated for each submodule. Finally, the first module may be transformed, using automatic transformations if available, into function form. Alternatively, all modules may be transformed into some other language, for example an imperative language, and tested using the generator version.

A well-known example, the Naur text formatter, is presented.

LOGIC PROGRAMMING — What does it bring to the software engineering?

Toshiaki Kurokoawa

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

In this paper are studied the impacts of logic programming on the software engineering field. Logic programming will solve many existing problems of the software written in conventional languages. However, new problems will be also brought by the logic programs. It is especially true when you use the actually implemented version of a logic programming language such as PROLOG.

A conclusion is given that logic programming is a new discipline and a new way of software production. And the experience of software engineering can contribute to the creation of the really useful programming language based on logic. Several attempts on the development of a logic programming language are surveyed to investigate the implication to and from the software engineering activities.

A Design Methodology in Prolog Programming

Z. Markusz

Computer and Automation Institute,
Budapest, Hungary

A. A. Kaposi

Polytechnic of the South Bank, London, UK

Complexity control has been extensively used as an aid in developing CAD programs for various architectural, mechanical and production engineering applications.

This paper outlines the process of evolving a complexity controlling design methodology. It discusses the experience with its use and analyses the complexity properties of programs designed by different methods.

Complexity control may also be achieved retrospectively, or in the course of maintenance, by measuring the complexity of a program, identifying its most complex parts and reconstructing these as structures of simple components. This procedure may offer opportunities to identify re-usable program modules, thus leading to elegant program designs.

Towards a Derivation Editor

Agneta Eriksson, Anna-Lena Johansson and *Sten-Åke Tärnlund*

UPMAIL
Computing Science Department
Uppsala University, Sweden

We shall take up a derivation editor for four classes of derivations of logic programs. The purpose of an editor is to support an interactive construction of derivations so the informal reasoning of the user can be carried over to the formal reasoning of the editor and conversely. A few implemented editors for logic programs have in particular, made it possible to formally prove a correctness theorem of an insert program on ordered binary trees. No automatic theorem prover is probably able to prove this theorem to-day, and may be a person would not do it either.

Alternation and the Computational Complexity of Logic Programs

Ehud Y. Shapiro

Department of Computer Science
Yale University
New Haven, CT 06520, USA

We investigate the complexity of derivations from logic programs, and find it closely related to the complexity of computations of alternating Turing machines. In particular, we define three complexity measures over logic programs — goal-size, length and depth — and show that goal-size is linearly related to alternating space, the product of length and goal-size is linearly related to alternating tree-size, and the product of depth and goal-size is linearly related to alternating time. The bounds obtained are simultaneous.

As an application, we obtain a syntatic characterization of Nondeterministic Linear Space and Alternating Linear Space via logic programs.

The Undecidability of two Completeness Notions for the «Negation as Failure» Rule in Logic Programming

Howard A. Blair

Computer Science Department
Iowa State University, USA

Two foundational results concerning completeness notions for the «negation-as-failure» (NAF) rule for logic programs are reported. Two natural notions of completeness of the «negation-as-failure» rule arise. First, given a logic program P, and a predicate symbol A, is it the case that for every ground atom A' with A as predicate for which not-|— pA' holds, that not-|— pA' is indeed provable? (NC1) Second, is it the case that for every ground atom A for which -A is Herbrand valid in logic program P, that -A is provable in P by the «negation-as-failure» rule? (NC2)

The first of these two notions is of general interest and is independent of the closed world assumption. The decision problem for the completeness of NAF for (P,A) under (NC1) is Π_2-complete. The decision problem for the completeness of NAF for P under (NC2) is Π_3-complete. The results are derived by reducing (NC1) and (NC2) to recursion-theoretic notions via fixed-point semantics for logic programs.

abstracts

Negation and Semantics of Prolog Programs

Taisuke Sato

Electrotechnical Laboratory
1-1-4, Umezono, Sakura-mura
Nühari-gun, Ibaraki, Japan

The semantics of Prolog programs with 'not' is discussed. It is shown that whereas a Prolog program with 'not' sometimes fails to have a minimum model which corresponds to the positive CWA (Closed World Assumption) it always has an N model irrespective of 'not'. N∝ model proposed here corresponds to the negative CWA. Since both models agree on the truth value of P as far as call(P) terminates, the N∝ model can be seen as a natural extension of the minimum model. Based on the N∝ model, it is also shown that «Negation as Failure» inference rule is sound as far as the ground condition, P is ground whenever not(P) is called, is kept.

Towards an Inductionless Technique for Proving Properties of Logic Programs

Roberto Barbuti, Pierpaolo Degano and Giorgio Levi

Istituto di Scienze dell'Informazione,
Università di Pisa, Italia

The paper presents a non-inductive method for proving properties of logic programs, based on the Knuth-Bendix completion algorithm. The language we use to express programs is PROLOG, constrained by distinguishing inputs and outputs, and by imposing conditions which forbid to write non-deterministic programs. In order to naturally express program properties, PROLOG has been extended to allow clauses having more than one atom in their left-hand side, and the computation rule is accordingly extended. An example is presented to illustrate the power of the method.

Unification-based Conditional Binding Constructs

Harvey Abramson

Department of Computer Science
University of British Columbia
Vancouver, Canada

The unification algorithm, heretofore used primarily in the mechanization of logic, can be used in applicative programming languages as a pattern matching tool. Using SASL (St. Andrews Static Language) as a typical applicative programming language, we introduce several unification based conditional binding (ie, pattern matching) constructs and show how these can promote clarity and conciseness of expression in applicative languages, and we also indicate some applications of these constructs. In particular, we present an interpreter for SASL functions defined by recursion equations.

Relational Production Systems and Logic Programs

Paul H. Morris

University of California,
Irvine, California, USA

The reletionship of logic programs to problem solving systems is investigated further. The latter are represented by a version of the relational production systems of Vere, modified to use multisets rather than sets. It is shown that a Horn clause logic program may be precisely modeled by a relational production system with empty initial state, leading to an explanation of why a strong form of goal conflict is not an issue for logic programs. The model provides a type of operational semantics for the top-down interpreter. This is illustrated with the Fibonacci example. The results suggest that certain logically equivalent formules should be given distinct procedural interpretations. Implications of the multiset formalism for plan formation using logic are briefly discussed.

A Comparison of the Logic Programming Language Prolog with Two-Level Grammars

Jan Maluszyński

Software Systems Research Center
University of Linköping, Sweden

A comparison of Horn clause logic programming systems and two-level grammars is carried out. The comparison is mainly informal, based on a language analysis example. It is described how two-level grammars may be imposed restrictions enabling their use as a computational tool similar to Prolog. Moreover it is explained how the language generation feature of two-level grammars might serve as a means of controlling the nondeterministic computation in order to reduce backtracking.

Medical Decision Aid: Logic Bases of the Sphinx System

M. Joubert, M. Fieschi, D. Fieschi, M. Roux

Service Universitaire de Biomathématiques,
Statistiques Médicales et Epidémiologiques,
Informatique, Faculté de Médecine
de Marseille, France

The basic elements of the SPHINX interactive system medical decision aid are here represented and discussed. They are the literals of a typed logic of which we present the particularities. They make it possible to express specialised knowledge in the form of independent rules. In the interpretation of the latter, the usual unification is replaced by a unification which takes into account the implicit properties of the types according to their particular structure. Examples of knowledge expression are presented, and the way in which knowledge is used by the man-machine dialogue procedure, as well as by the decision procedure.

A Dialogue in Natural Language

Robert Pasero

Groupe d'Intelligence Artificielle
Faculté des Sciences de Luminy
Marseille, France

In this paper we present a way of constructing a dialogue between man and machine, through a simple syntactic and semantic system. We also give a Prolog program as a core for a more sophisticated system.

First we describe a dialogue's language syntactic structure, then we define a three-valued logic system in order to represent the semantics of this language. Afterwards, we simulate this three-valued logic as a binary logic, for the purpose of the program.

abstracts

Application of Meta Level Programming to Fault Finding in Logic Circuits

Kave Eshghi

Department of Computing Imperial College of Science and Technology
London, UK

FAULTFINDER is a program that diagnoses faults in digital circuit modules. It relies on the input-output behaviour of the circuit to achieve this, without requiring access to the circuit elements directly. It uses techniques of meta language programming in logic.

Dado: a Tree-Structured Machine Architecture for Production Systems

Salvatore J. Stolfo and *David E. Shaw*

Department of Computer Science
Columbia University
406 S. W. Mudd
New York, NY 10027 USA

Dado is a parallel, tree-structured machine designed to provide highly significant performance improvements in the execution of production systems.

Exponential Improvement of Exhaustive Backtracking: Data Structure and Implementation

Stanislaw Matwin

Department of Computer Science
University of Ottawa
Ottawa, Ontario, Canada

Tomasz Pietrzykowski

School of Computer Science
Acadia University
Wolfville, Nova Scotia, Canada

The paper presents the data structure enabling an implementation of an efficient backtracking method for plan-based deduction. The structure consists of two parts. Static structure combines information about the initial set of clauses and unifiers. Dynamic structure records information about the history of the deduction process (plan graph), about the unifications built in this process (graph of constraints), and about the conflicts encountered. Dynamic structure consists purely of pointers and integers and is extremely economical.

The system operates on units. The units describe attempts to find the solution and reside on disk. A procedure controlling traffic of units between disk and memory and guaranteeing completeness of the search is presented. Some other procedures (development and pruning of plans, development of constraints graph and conflict detection) are also discussed.

The last section presents an example of a problem and follows its solution by a conventional PROLOG interpreter and by our system.

An Introduction to Mu-Prolog

Lee Naish

Department of Computer Science
University of Melbourne
Parkville, Victoria 3052 Australia

As a logic programming language, PROLOG is deficient in two areas: negation and control facilities. Incorrectly implemented negation affects the correctness of programs and poor control facilities affect the termination and efficiency. These problems are illustrated by examples.

MU-PROLOG in then introduced. It correctly implements negation and has more control facilities. Control information can be added automatically. This can be used to avoid infinite loops and find efficient algorithms from simple logic. MU-PROLOG is closer to the ideal of logic programming.

Foundations of Logic Programming

J. W. Lloyd

Department of Computer Science
University of Melbourne
Parkville, Victoria 3052 Australia

These notes given an account of the mathematical foundations of logic programming. It is assumed that the reader is familiar with a logic programming language, such as PROLOG, and hence appreciates the interest in the results presented here.

Chapters 1 to 3 contain the most fundamental results concerning soundness, completeness and the negation as failure rule. Most of these results are well known, although many of the proofs are new. Further chapters on more advanced topics are planned.

MU-PROLOG 2.4 Reference Manual

Lee Naish

Department of Computer Science
University of Melbourne
Parkville, Victoria 3052 Australia

The MU-PROLOG interpreter is written in C and is especially suited to Unix environments. MU-PROLOG is intended to be upward compatible with DEC-10 PROLOG and Unix PROLOG. The syntax and built-in predicates are therefore very similar. MU-PROLOG correctly implements negation and has more control facilities than standard PROLOGs.

Chaotic Semantics of Programming Logic

J-L. Lassez and *M. J. Maher*

Department of Computer Science
University of Melbourne
Parkville, Victoria 3052, Australia

We use the notions of closures and fair chaotic iterations to give a semantic to logic programs. The relationships between the semantics of individual rules and the semantics of the whole program are established and an application to parallel processing is mentioned. A chaotic fixed point theorem is given, which carries the non-determism inherent to resolution. Finally, a result of Apt and van Emden is reinterpreted by establishing the soundness and completeness of fair SLD resolution with respect to finite failure.

Programming Law in Logic

Allen Hustler

Department of Computer Science
University of Waterloo, Canada

This paper briefly reviews recent work on automating the legal research process; investigates the relationship between law and logic, and the work done by legal academics in this area; and describes a prototype interactive logic-based legal consultant written in the PROLOG language.

abstracts

Logic Programming for Expert Systems

Peter Hammond

Department of Computing
Imperial College, 180 Queen's Gate
London SW7 2BZ, UK

The expert System MYCIN and PROSPECTOR an discussed in detail. A PROLOG implementation MYCIN (PROLOG) is described and a faultfinding system FAULTFINDER, also implemented in PROLOG, is defined.

Knowledge-Based Retrieval on a Relational Database Machine

David Elliot Shaw

Department of Computer Science
Columbia University
406 S. W. Mudd
New York, NY 10027 USA

The central focus of this research has been the efficient retrieval of records from very large databases in applications where the criteria for description-matching require deductive inference over a domain-specific «knowledge base». Our approach has involved the design of a specialized non-von Neumann machine which permits the highly efficient evaluation of certain operators of a *relational algebra* of particular importance to the computational task of logical satisfaction. The architecture permits an *O(log n)* improvement over the best known evaluation methods for these operators on a conventional computer system, and may also offer a significant improvement over the performance of previously implemented or proposed database machines in other applications of pratical import.

On Determining the Causes of Nonunifiability

P. T. Cox

Department of Computer Science
University of Auckland
Auckland, New Zeland

Determining whether or not a pair of functional expressions is unifiable is a problem of major importance in many processes, mechanical theorem-proving in particular. As a result, the unification problem has been intensively studied. A related problem of equal importance is how to deal with unification failure. Most processes using unification perform some search, and must backtrack when unification fails. The usual strategy is to backtrack to the last point at which unification was successful. This may not be the correct place to resume the search, however, and although the correct place will eventually be found, irrelevant areas of the search space will be investigated first. As a step towards more intelligent backtracking behaviour, we present a method for determining *all* the causes of nonunifiability.

Deduction Plans: A Basis for Intelligent Backtracking

Philip T. Cox

Department of Computer Science
Auckland University
Auckland, Australia

Tomasz Pietrzykowski

School of Computer Science
Acadia University
Wolfville, N. S., Canada

A proof procedure is described that relies on the construction of certain directed graphs called «deduction plans». Plans represent the structure of proofs in such a way that problem reduction may be used without imposing any ordering on the solution of subproblems, as required by other systems. The structure also allows access to all clauses deduced in the course of a proof, which may then be used as lemmas. Economy of representation is the maximum attainable, consistent with this unrestricted availability of lemmas

Restricted versions of this deduction system correspond to existing linear deduction procedures, but do not suffer from some of their shortcomings, such as redundant representation, strict ordering of subproblems, and explicit substitution. One of the rules for constructing plans, however, allows a subproblem to be factored to a previously solved one: this has no equivalent in existing systems.

A further economy is obtained by making it unnecessary to perform substitutions and calculate most general unifiers.

The source of unification failure can be located when a subproblem is found to be unsolvable, so that exact backtracking can be performed rather than the blind backtracking performed by existing systems. Therefore, a deduction system based on the construction of plans can avoid the wasteful search of irrelevant areas of the search space that results from the usual backtracking methods. Furthermore, because of the graphical structure, it is necessary to remove only the offending parts of the proof when a plan is pruned after backtracking, rather than the entire proof constructed after the cutting point.

The Database as Model: a Metatheoretic approach

Kurt Knolodige

SRI
Menlo Park, CA 94025, USA

This paper presents a method of formally representing the information that is available to a user of a relational database. The intended application area is deductive question answering systems that must access an existing relational database. To respond intelligently to user inquiries, such systems must have a more complete representation of the domain of discourse than is generally available in the tuple sets of a relational database. Given this more expressive representation, the problem then arises of how to reconcile the information present in the database with the domain representation, so that database queries can be derived to answer the user's inquiries. Here we take the formal approach of describing a relational database as the model of a first-order language. Another first-order language, the metalanguage, is used both to represent the domain of discourse, and to describe the relationship of the database to the domain. The formal advantages of this approach are presented and contrasted with other work in the area.

Prolog Interpreter and its Parallel Extension

Yuji Matsumoto Katsumi Nitta

Electrotechnical Laboratory
Ibaraki, Japan, 305

Koichi Furukawa

Institute for New Generation
Computer Technology
Tokyo, Japan, 108

This paper presents a Prolog interpreter based on concurrent programming and its

extension to a parallel execution model. The execution of a Prolog program can be seen as a depth first traverse of an AND/OR tree defined by the program. Our interpreter is based on this point of view, and consists of two kinds of processes, AND-process an OR-process.

Exponential Improvement of Exhaustive Backtracking: A Strategy for Plan-Based Deduction

Tomasz Pietrzykowski

School of Computer Science
Acadia University
Wolfville, Nova Scotia, Canada

Stanislaw Matwin

Department of Computer Science
University of Ottawa
Ottawa, Ontario, Canada

The paper presents a method of mechanical deduction. Attempts to find refutation(s) are recorded in the form of triples: plan, constraints, conflicts. A plan corresponds to a portion of AND/OR graph search space and represents purely deductive structure of derivation. Constraints form a graph recording the attempts of unification, while conflicts identify minimal subset of the plan, removal of which restores unifiability.

This method can be applied to any initial base of (nonnecessarily Horn) clauses. Unlike the exhautive (blind) backtracking which treats all the goals deduced in the course of proof as equally probable source of failure, this approach detects the exact source of failure.

In this method only a small fragment of solution space is kept on disk ss a collection of triples. The search strategy and the method of non-redundant processing of individual triples which leads to a solution (if it exists) is presented. This approach is compared — on a special case — with blind backtracking and an exponential improvement is demonstrated.

Algorithmic Program Debugging

Ehud Y. Shapiro

Department of Computer Science
Yale University, USA
(Thesis)

In this thesis we lay a theoretical framework for program debugging, with the goal of partly mechanizing this activity. In particular, we formalize and develop algorithmic solutions to the following two questions:

1. *How do we identify a bug in a program that behaves incorrectly?*
2. *How do we fix a bug, once one is identified?*

We develop interactive diagnosis algorithms that identify a bug in a program that behaves incorrectly, and implement them in Prolog for the diagnosis of Prolog programs. Their performance suggests that they can be the backbone of debugging aids that go far beyond what is offered by current programming environments.

We develop an inductive inference algorithm that synthesizes logic programs from examples of their behavior. The algorithm incorporates the diagnosis algorithms as a component. It is incremental, and progresses by debugging a program with respect to the examples. The Model Inference System is a Prolog implementation of the algorithm. Its range of applications and efficiency is comparable to existing systems for program synthesis from examples and grammatical inference.

We develop an algorithm that can fix a bug that has been identified, and integrate it with the diagnosis algorithms to form an interactive debugging system. By restricting the class of bugs we attempt to correct, the system can debug programs that are too complex for the Model Inference System to synthesize.

Extended Integrity Constraints in Query-by-Example

J. C. Neves, M. H. Williams and *S. O. Anderson*

Computer Science Department,
Heriot-Watt University
Edinburgh EH1 2HJ, UK

The specification and enforcement of integrity constraints has become an important aspect of data base systems, and data base management languages must make provision for them. In the case of the data base language Query-by-Example a style for handling certain types of integrity constraints has been developed by Zloof. An alternative approach is presented here which allows for a more general type of constraint. This includes the handling of functional, multivalued and embedded-multivalued dependencies and the more conventional and simpler type of integrity contraint in a uniform manner.

A Prolog Implementation of Query-by-Example

J. C. Neves, R. C. Backhouse, S. O. Anderson and *M. H. Williams*

Computer Science Department,
Heriot-Watt University
Edinburgh EH1 2HJ, UK

Prolog is a programming language which is particularly well suited to data base applications. However, since formulating queries for Prolog data bases requires a knowledge of Prolog syntax, various workers have been involved in developing better interfaces to Prolog data bases. Query-by-Example (QBE) is a data base management language which provides a simple interface to relational data bases and which has a syntax which is very similar to Prolog goals. This paper examines the correspondence between QBE and Prolog from the viewpoint of a particular implementation, and concludes that QBE is a natural choice as an interface to Prolog data bases.

Designing Transparent Natural Language Interfaces for Information Systems

Paul Sabatier

Laboratoire d'Automatique
Documentaire et Linguistique
Université de Paris 7
2, Place Jussieu
75221 Paris Cedex 05, France

Natural language interfaces would gain in convenience and interest if the user were able to know what the system can understand, i.e., if they could reveal to him their linguistic capabilities and their limits, thus becoming transparent. This can be achieved indirectely, by producing explanatory and talkative messages when the analysis of a sentence fails; and directly, by allowing the user himself to ask the interface questions about its linguistic competence. We discuss these two aspects and their consequences for the design of interface involving such a capability of transparence.

---

research centres addresses

Here are some more:

Dept. of Computer Science
University of New South Wales
P.O.B. 1, Kensington, NSW 2033 Australia

Dept. of Computer Science
University of Melbourne
Parkville, Victoria 3052 Australia

Imecc_Unicamp cp 1170
Campinas SP 13100 Brasil

Dept. of Computer Science
University of Ottawa
Ottawa, Ontario K1N 9B4 Canada

School of Computer Science
Acadia University
Wolfville B0P 1X0 Canada

Institute for Application of
Computing Technique in Control
Revolucni 24, Praha 1 Czechoslovakia

DIKU
Sigurdsgade 41
DK- 2200 Copenhague N Denmark

Institut für Informatik
Technische Universität München
Postfach 202420
D-8000 Munchen 2 Deutschland

Imag, Université de Grenoble I
B. P. 53X, 38041 Grenoble Cedex France

CRISS
BP 47X
38040 Grenoble Cedex France

Service Universitaire de Biomathématique
Informatique, 27 Bd. Jean Moulin
13385 Marseille Cedex 5 France

Atelier Microinformatique
Campus Universitaire de Beaulieu
Avenue Du General Leclerc
35042 Rennes Cedex France

CNET LAA/SLC/PGL
BP 40, 22301 Lannion Cedex France

LISH
Université de Toulouse le Mirail
109 bis rue Vouquelin
31058 Toulouse Cedex France

LSI
Université Paul Sabatier
118 Route de Narbonne
31062 Toulouse Cedex France

Ensae-Cert
Complexe Aerospatial de Lespinet
10 Av. Edouard Belin
31055 Toulouse Cedex France

Centre Mondeal
22, Av. Matignon
75008 Paris France

Laboratoire d'Informatique
École Normale Supérieure
5, Rue Marie Davy
75014 Paris France

Otszk
P. O. B. 161
Budapest H-1440 Hungary

Chinoin
P. O. B. 110
Budapest H-1325 Hungary

Vidioton
P. O. B. 65
Budapest H-1525 Hungary

Kfki
P. O. B. 49
Budapest H-1525 Hungary

Institute for Coordination
of Computer Techniques
1054 Budapest, Akademia UTCA 17
Hungary

Dept. of Applied Mathematics
Weizmann Institute
Rehovot 7600 Israel

Information Engineering Course
University of Tokyo, Graduate School
Bunkyo, Honso
Tokyo 113 Japan

ICOT — Institute for New Generation
Computer Technology
Mita Kokusai Building, 21F
4-28, Mita 1-Chome, Minato-Ku
Tokyo 108 Japan

Dept. of Computer Science
University of Auckland New Zealand

Runit Computing Centre
7034 Trondheim-NTH Norway

Dept. of Computer Systems
The Royal Institute of Technology
S-100 44 Stockholm 70 Sweden

Cognitive Studies Programme
Arts Building
University of Sussex
Brighton BN1 9QN UK

Dept. of Computer Science
Herriot-Watt University
Grassmarket, Edinburgh UK

research centres addresses

Dept. of Computer Science
University of Exeter
Stocker Road, Exeter EX4 4QL **UK**

Robot Welding Project
Dept. of Engineering Science
University of Oxford, Parks Road
Oxford OX1 3PJ **UK**

Expert Systems Ltd.
34 Alexandra Road
Oxford OX2 0DB **UK**

Dept. of Computer Science
University of Delaware
Newark DE 19711 **USA**

IBM Corporation
P. O. Box 390
Poughkeepsie
NY 12602 **USA**

Burroughs Corporation
Federal and Special Systems Group
P. O. Box 517
Paoli PA 19301 **USA**

Dept. of Computer Science
Virginia Polytechnic Institute
Blacksburg VA 24061 **USA**

Centro de Computación
Universidad Simon Bolivar
Caracas **Venezuela**

STOP PRESS

Digital

Information and reports about Prolog implementation techniques and applications are wanted at Digital. Mail them to:

Michael Poe
Mailstop LTN1 — 2/HO4
295 Foster St.
Littleton, Mass. 01460, USA

First Announcement

The János Bolyai Mathematical Society intends to organize a "Colloquium on Algebra, Combinatorics and Logic in Computer Science" in the period 12-16, September 1983 at Györ, Hungary.

The aim of the colloquium is to provide ground for the exchange of information on new achievements and the recent problems in any field of algebra, combinatorics and logic related to: the theory of automata, tree automata, computation and languages, semantics, complexity, computability, decidability, data structures, data manipulation and data analysis.

Other fields of computer science applying algebra, combinatorics and logic are also welcome.

We intend to publish the Proceedings of the Colloquium in a volume of the series Colloquia Mathematica Societatis J. Bolyai published jointly by the Society and North-Holland, Amsterdam.

Interested colleagues are kindly requested to contact the secretary:

B. Uhrin, János Bolyai Mathematical Society,
1061 Budapest, Anker köz 1-3. Hungary.