UPMAIL Technical Report No. 30

# LM-Prolog
# The Language and Its Implementation

*Mats Carlsson*

**March 29, 1986**

UPMAIL
Uppsala Programming Methodology and Artificial Intelligence Laboratory
Department of Computing Science, Uppsala University
P.O. Box 2059
S-750 02 Uppsala Sweden
Domestic 018 155400 ext. 1860
International +46 18 111925
Telex 76024 univups s

This report was written in partial fulfillment of the requirements for the degree of Licentiat of Philosophy in Computing Science at Uppsala University.

The research reported herein was sponsored by the National Swedish Board for Technical Development (STU).

**Keywords:** Prolog, Logic Programming, Unification, Lisp, Functional Programming, Continuations, Compiling

## Abstract

We have implemented logic programming in a functional setting. Continuations are used as a technique for making procedures of the implementation language compute relations rather than functions.

A new technique based on a 'genetic code' for terms is used for constructing complex terms in a demand-driven fashion during unification. A microcoded unifier using dispatch hardware makes the genetic code technique feasible. The technique has been generalized to handle variable annotations.

Our language, LM-Prolog, includes significant extensions of the Prolog language, e.g. demand-driven, data-flow, and parallel computations, multiple databases, user extensible unification, optional prevention or handling of cyclic terms, and the ability to call Lisp functions. Terms of LM-Prolog are built up from variables, atoms, and binary records (conses).

The implementation consists of an interpreter and a compiler. The compiler transforms LM-Prolog predicates to Lisp procedures which are translated into an extended machine instruction set by the host machine's Lisp compiler.

LM-Prolog is implemented on MIT Lisp Machines, which provide a very rich environment for developing and running programs. They are personal work stations with large address spaces, large amounts of memory and disk storage, and high resolution graphics. A high-performance implementation of a logic programming language is for the first time available in such a powerful programming environment.

# 1. Introduction

## 1.1 Overview

LM-Prolog is an implementation of a logic programming language on MIT Lisp Machines [Greenblatt, 1974]. This thesis is a description of LM-Prolog, focusing on its implementation. A description from the user's point of view can be found in [Carlsson and Kahn, 1983].

Throughout, **boldface** stands for LM-Prolog entities; *italics* introduces new concepts and is used for syntactic variables; $\overline{u}$ denotes the result of compiling $u$.

This chapter introduces the language and the principal design decisions underlying its implementation, and gives pointers to other chapters that go into details. Chapter 3 gives a general overview of the implementation.

## 1.2 Language and Computational Model

Our language is essentially the Horn clause formalism for logic programming [Kowalski, 1974]. A program is composed of a set of predicate [1] definitions, each consisting of a sequence of Horn clauses together with optional pragmatic information. The clauses do not have to be strictly first-order, but may contain meta-logical goals, control goals and *collectors*. A collector constructs the list of all members of a relation. In this thesis, we call such lists *collections* rather than sets, since they do not have all the properties of sets. The idea of having collectors in a logic programming language originates from IC-Prolog [Clark *et al.*, 1982].

The terms of LM-Prolog are built up from variables, atoms, and binary records (conses). We take the approach of LogLisp [Robinson and Sibert, 1980] and use the pair constructor as the only constructor. The terms are implemented as Lisp Machine objects; in particular, binary records map to Lisp *conses* and variables map to Lisp *locatives*. Everything else is an atom.

The computational model of LM-Prolog is by default the fixed sequential left-to-right, depth-first execution of goals and sequential consideration of alternative clauses by backtracking. This basic computational model, which is the standard one for any Prolog implementation, can however by the use of special annotations be replaced by and intermixed with alternative models such as

- demand-driven computation via lazy collectors which find elements of a relation as they are needed (driven by unification), rather than all elements at once;
- data-flow computation via *constraints*, offering a means for restricting the range of variables by associating an uninstantiated variable with a goal to be executed when the variable is instantiated. It is remarkable that a class of "generate and test" programs can be written both efficiently and clearly using this technique;
- parallel processing via eager collectors which spawn a separate process so that the collection is computed in parallel with its consumer (synchronized by unification);

[1] In this thesis, we do not keep a strict terminological difference between "a predicate of some arguments" and "a relation over some domains"

- functions, by means of a mechanism for calling Lisp functions with negligible overhead;
- data-flow synchronization in the context of and-parallelism by means of variable annotations [Clark *et al.*, 1982]. This mechanism was added to the language for supporting an interpreter for the logic programming language Concurrent Prolog [Shapiro, 1983].

These computational models and their implementation are described in Chapter 8. They all capitalize on the fact that our unifier is *open-ended* so that new datatypes and the semantics for their unification with other objects can be added in a modular fashion. For efficiency, our unifier is by default ignorant of the fact that cyclic terms may result from unification. It is well known that such terms jeopardize the consistency of the logic system, and unification itself may not terminate. Therefore, users who are willing to pay a certain runtime penalty can optionally tell the unifier to prevent cyclic structures from occurring. It is also possible to tell the unifier, still at a certain penalty, to allow cyclic structures while retaining its termination properties. More on unification in Chapter 5.

LM-Prolog's database is structured into collections, called *worlds* [Nakashima, 1982], of predicate definitions. The search order of the database is defined by the *universe* which is a list of worlds. Worlds are generally useful for structuring of knowledge and modularity. Programs may dynamically add and delete clauses from worlds, and also add and delete worlds from the universe. Chapter 4 contains a detailed description of the database.

A predicate can optionally be declared to be deterministic, i.e. that any invocation of it shall have at most one solution. Determinism, either declared or detected at compile time, plays an important rôle in the tasks of efficient interpretation and generation of efficient code. The interpretation and compilation processes are described in Chapter 6 and 7.

Other declarable pragmatic properties of predicates include indexing, assertability, compiled or interpreted execution mode, world membership, and on-line documentation. The language description is continued in Chapter 2.

## 1.3 Construction of Complex Terms

### 1.3.1 Structure Sharing (SS)

In some Prolog implementations including the first [Battani and Meloni, 1973] and DEC-10 Prolog [Warren, 1977], the representation of a term is a variant of the elegant *structure sharing* technique [Boyer and Moore, 1972]. A term is represented by two pointers, one to the pure code, the other to a binding environment containing the bindings of the variables in the pure code. A variable in the pure code is represented by a distinct type marker and an integer offset. In structure sharing implementations, then, every part of the abstract machine executing Prolog must deal with *molecules* i.e. pairs of pointers representing terms.

Structure sharing is particularly advantageous on machines whose instruction set support user-defined base registers. The environment part of a molecule can then reside in a base register, and a variable is then looked up by offsetting from the base register. This is heavily exploited in DEC-10 Prolog.

### 1.3.2 Copying Pure Code (CPC)

As an alternative to structure sharing, terms can be represented directly, without need for an additional pointer to a binding environment. Whenever a free variable is unified with a term of pure code containing variables, a copy of the term is made with the variables replaced by their bindings if they are bound, or by newly allocated free variables if they are unbound.

Bruynooghe labeled this implementation technique *copying pure code*, see [Bruynooghe, 1982a].

Of course, the terms are not quite represented directly, since the bindings of variables inside complex terms still have to be looked up. Because of the copying scheme, however, there are much fewer such variables than in the structure sharing case. Copying can be avoided in the case of variable free terms, since variable free terms of pure code can be represented directly and be structure shared in the sense of having several pointers to them.

### 1.3.3 Invisible Pointers

We have chosen the CPC approach, partly because it happens to map extremely well to the Lisp Machine. Using a kind of forwarding pointer datatype called *invisible pointer*, terms can really be represented directly in the normal sense of the word. The look-up of variables inside complex terms is handled by the firmware, and so the variables become invisible to all normal uses of the terms. In particular, interfacing to Lisp functions, passing terms as Lisp objects without any conversion or other overhead, becomes feasible. Details of the invisible pointer mechanism are discussed in Section 5.2.

Other factors advocating the choice of CPC rather than SS on the Lisp Machine include:

- The machine cannot hold two addresses in one word.
  In [Mellish, 1982], this is claimed to be a significant factor in the choice, since the necessity to store a 'molecule' in two machine words leads to twice as many memory accesses for looking up the value of a variable.
- By the same token, a logical variable would have to map to two local Lisp variables in compiled predicates, causing the stack to grow almost twice as fast as in the present CPC implementation.
- The machine's instruction set does not support user-defined base registers and it seems that a dedicated instruction for doing so would incur too large an overhead.

### 1.3.4 Object and Source Terms

For the purposes of this thesis, Prolog terms are called *object terms*, and their pure code representations are called *source terms*. Object terms are terms of the object language i.e. terms that normal programs compute with. Source terms are descriptions of object terms. A source term can unify with an object term. A source term can also be used to construct instances of the object term which it describes. Source terms are structure shared constants with special annotations, principally for first and single occurrences of variables and for ground subparts of the terms they describe.

There are two ways for object terms to be constructed:

- When a variable is matched against a source term, the variable is bound to an object term constructed out of the source term.
- Object terms are also constructed as arguments in procedure calls.

## 1.4 Logic Programming in a Functional Setting

Logic programs are based on relations and functional programs are based on functions. Since functions are a special case of relations, it would seem that functional programming could easily be implemented in a logic programming setting (and this is indeed so), but not necessarily vice versa.

One of the results of this thesis is a technique for making procedures of a functional language compute relations rather than functions. The technique is called *continuation passing* and is the workhorse for implementing Prolog relations as Lisp procedures.

An important feature of logic programming, extraneous to functional programming, is the logic variable and its incorporation in complex terms. It would seem that logic programming terms must be incompatible with the data structures of functional programs. Luckily, due to the invisible pointer mechanism discussed in Sections 1.3 and 5.2, this incompatibility does not occur in the present implementation, and Prolog terms can be represented directly as their equivalent Lisp terms. Therefore, the interface between Lisp and Prolog becomes extremely simple and cheap.

### 1.4.1 Continuation Passing

*Continuations* [Strachey and Wadsworth, 1974] are well known from denotational semantics theory, and were first conceived as a formalization of the semantics of jumps in imperative programming languages. Later, they have been used to characterize non-deterministic constructs in functional languages. These techniques are exemplified in [Henderson, 1980].

Continuation passing consists in passing each procedure as an extra argument a function of no arguments. This function has typically been created by another procedure closing over some of its local variables. The continuation describes the computation that remains to be done once the called procedure has finished *its* work, at which point the continuation is *invoked* i.e. applied as a function of no arguments. Once the continuation has terminated, the called procedure may find an alternative solution and may re-invoke the continuation, and so on.

For example, the relation $p$ defined as

$$p(X) \leftarrow q(X) \wedge r(X).$$
$$p(2).$$

could be translated into the procedure

```
(defun p' (cont x)
  (or (q' #'(lambda () (r' cont x)))
      (unify cont x 2)))
```

where unify is a function of a continuation and two terms which invokes the continuation with the two terms unified. In the example, the procedure $p'$ passes a continuation closing over cont and x to $p'$. When invoked, this continuation will pass the original continuation cont to $r'$. If it ultimately fails, a unification of $x$ and 2 will be attempted and, if successful, cont will be invoked. (Details of Lisp Machine Lisp syntax are given in Section 2.2.)

The potential of continuation passing is capitalized in our implementation as follows: The satisfaction of a conjunction $P \wedge Q$ corresponds to a call to a procedure representing $P$

with an extra continuation argument representing $Q$. If the procedure succeeds in finding a member of the relation it computes, it invokes its continuation. If the value eventually returned by the continuation is nil, the procedure goes on to find more solutions, i.e. backtracking is triggered, and more members of the relation can be found.

The continuation passing mechanism has been exploited to enforce determinacy, to implement collectors, and more, in addition to achieving backtracking. In addition to being passed as arguments, continuations can also be stored away in data structures. See Section 5.3.

Stepwise refinement of Prolog interpreters using the mechanism can be found in [Carlsson, 1984].

### 1.4.2 Realization of Continuations

A continuation is realized as a structure which contains a function and a set of arguments for that function. Invoking a continuation consists in applying the function part to the arguments.

# 2. The LM-Prolog language

## 2.1 Introduction

LM-Prolog stands for *Lisp Machine Prolog.* It is a Prolog, although with an extended control language, and it is thoroughly embedded in the programming environment of personal Lisp Machines.

The data structures are based on lists as opposed to records, and so is the syntax, which is very close to that of LogLisp [Robinson and Sibert, 1980] or QLOG [Komorowski, 1980]. We do not support infix operators, as opposed to micro-Prolog [Clark and McCabe, 1984].

As an example of LM-Prolog, consider the following definition of some family relations.

```
Logic                                      LM-Prolog

                                           (define-predicate grandparent
grandparent(Gp, Gc) ←                        ((grandparent ?gp ?gc)
 parent(P, Gc)∧                               (parent ?p ?gc)
 parent(Gp, P).                               (parent ?gp ?p)))

                                           (define-predicate parent
parent(P, C) ←                               ((parent ?p ?c)
 father(P, C).                                (father ?p ?c))
parent(P, C) ←                               ((parent ?p ?c)
 mother(P, C).                                (mother ?p ?c)))

                                           (define-predicate father
                                             (:options (:type :dynamic))
father(GustafAdolf, CarlGustaf).             ((father Gustaf-Adolf Carl-Gustaf))
father(CarlGustaf, Victoria).                ((father Carl-Gustaf Victoria))
father(CarlGustaf, CarlPhilip).              ((father Carl-Gustaf Carl-Philip))
father(CarlGustaf, Madeleine).               ((father Carl-Gustaf Madeleine)))
```

The principal means of defining predicates is by using the define-predicate statement, which is both a Lisp special form and an LM-Prolog predicate so it can be entered at top-level to either system or included in Lisp files.

Its arguments are mainly the clauses of the predicate, written as lists

$$(P\ Q_1 \ldots Q_n)$$

to be read as "$P$ is true if each of $Q_1 \ldots Q_n$ is true".

By default, relations in the language are not mutable, but the declaration (:options (:type :dynamic)) informs LM-Prolog that programs are allowed to update the father relation. Other declarations can be given as :options sub-specifications. In particular, a predicate can be declared to be *deterministic* if it is desired that it should never try to find more than its first solution. This control annotation can be regarded as an alternative to the *cut* primitive ('!' or '/' in other references) and has the stylistic advantage of a clearer separation between logic and control. Indeed, it has been shown in [Nilsson, 1984] that all uses of *cut* can be replaced by this control annotation.

LM-Prolog predicates may call Lisp functions. Although this involves two inference mechanisms, namely resolution for Prolog and a variant of lambda-calculus for Lisp, the semantics of function calls is exactly as if functions were a part of the logic programming language itself. A call to a Lisp function is recognized by a syntactical device, and is treated by the resolution machinery as an invocation of a deterministic predicate. The function eventually returns a value to be unified with a term specified together with the call.

LM-Prolog obeys the Lisp system's conventions regarding procedure calls and usage of the execution stack, and also inherits Lisp's datatypes and organizes its terms in such a way that they can be passed to Lisp directly without conversion. Therefore, calling Lisp functions from LM-Prolog incurs no overhead.

## 2.2 Lisp Machine Lisp

An accurate description of Lisp Machine Lisp can be found in [Moon, 1983].

### 2.2.1 Datatypes

Lisp objects are called *forms* and can be of two categories — *atoms* [1] and *conses* i.e. dotted pairs or binary records of any two forms called its *car* and *cdr*. A cons of $u$ and $v$ is abstractly written $u.v$.

The principal use of conses is for constructing *lists*. A list is recursively defined to be the empty list (abstractly written $\emptyset$), or a cons of an *element* and a list. In Lisp Machine Lisp, $\emptyset$ is written nil and $u_1.u_2\ldots u_n.\emptyset$ is written $(u_1\ u_2\ \ldots\ u_n)$, where $n \geq 0$. Non-lists of the form $u_1.u_2\ldots u_{n-1}.u_n$ are written $(u_1\ u_2\ \ldots\ u_{n-1}\ .\ u_n)$, where $u_n \neq \emptyset$.

Conses correspond to one particular Lisp Machine datatype, whereas atoms correspond to a multitude of datatypes, the principal ones being *symbols*, used as identifiers and written as any sequence of characters not confusable with some other object. Symbols may optionally have a *package prefix* specifying that the symbol belongs in a particular namespace. The package prefix is written as a sequence of characters followed by a colon ':'. Symbols used as *keywords* are written without prefix but with the colon, e.g. :handle.

Other atoms are *numbers* e.g. fixed point, arbitrary-precision, rational, floating point, or complex numbers, and *arrays*, which can be multi-dimensional and contain any terms. Many numerical array types are space optimized. *Strings* are a special case of arrays and can be read in as opposed to other array types. Strings are enclosed in double quotes '"'.

If strings are to contain double quotes, or if symbols are to contain certain special character, those characters have to be prefixed by a slash '/'. A symbol containing special characters may alternatively be enclosed in bars (| |).

A semicolon ';' indicates that the rest of the source code line is a comment, unless the ';' is protected by a slash or double quotes or bars.

We use a datatype called *locative* for the purpose of representing logic variables. Locatives are generally a kind of pure pointers to arbitrary memory locations. In this implementation, they are used in a restricted way which is described in Section 5.2.

[1] Not to be confused with atoms in resolution theory

### 2.2.2 Variables

Lisp Machine Lisp uses *local* and *special variables.* The scope of a local variables is restricted to the function defining it but does currently **not** include any internal functions of that function, i.e. (function (lambda ...)) constructs. Special variables are dynamically scoped.

### 2.2.3 Function definitions

The principal means of defining a Lisp function is by using the defun statement, used e.g. as

```
(defun fac (x)
  (cond ((zerop x) 1) (t (* x (fac (1- x))))))
```

The following syntactic variant is used for functions that take a variable number of arguments:

```
(defun how-many-args (&rest args) (length args))
```

An argument list of (x &rest y) means that the function takes at least one argument (x), and some additional arguments collectively viewed as the *&rest argument* y (a list).

### 2.2.4 Read macros

For convenience, the Lisp reader provides some shorthands for commonly used forms. In this thesis we will use '$x$, which is shorthand for (quote $x$), and #'$x$, which is shorthand for (function $x$). '$x$ evaluates to the form $x$ and #'$x$ evaluates to the function definition of the form $x$.

### 2.2.5 Flavors

An important programming subsystem of the Lisp Machine is the *flavor system.*

Flavors is the Lisp Machine subsystem for *object-oriented programming.* It is a sublanguage for defining abstract objects. Instantiations of these objects are called *instances.* The instances have two parts: *instance variables* and *methods.* Instance variables are variables that are local to the instance. The methods define the semantics of the abstract object. Flavors can also *inherit* all or some attributes from one or more superior flavors.

Specifying flavor methods is like specifying the operations of an abstract data type. An operation on a flavor instance is performed by *sending* it a *message.* This is equivalent to calling it as a function with the first argument a keyword symbol that selects the proper method. Whatever value the method returns is passed back.

A method is defined by the defmethod statement, e.g.

```
(defmethod (lazy-collection :unify) (other-term)
  (unify (send self ':ordinary-term) other-term))
```

This defines the :unify method of the lazy-collection flavor. The method takes one argument, other-term, and refers to the current instance (self).

## 2.3 LM-Prolog syntax and terminology

In LM-Prolog, we represent terms and all other logic programming constructs as Lisp forms, using Lisp Machine Lisp syntax except for logical variables, which are represented by locatives and written as variables beginning with '?'.

An LM-Prolog program is a set of *predicate definitions.* A definition consists of an *option spec* and a sequence of *clauses* pertaining to a particular predicate. The option spec is optional and contains on-line documentation, argument list declaration, database control, determinacy control, whether execution is compiled or interpreted, whether the predicate is *static* or *dynamic*, and more. Every definition goes into a *world* which is denoted by a symbol and may be given in the option spec. Worlds name collections of predicates and are discussed in the section about procedural semantics.

Clauses of dynamic predicates can be added or deleted (*asserted* or *retracted*) at run time, clauses of static predicates cannot.

Each clause comprises an optional *name*, a *head*, and a *body*. The name is just a symbol or omitted. Each definition is constrained to not contain two named clauses with identical names.

The body consists of a sequence of zero or more *predications* (called *goals*, *queries*, *literals*, or *boolean terms* in other references). A predication is a cons of a *predicator*, which names a predicate, and a list of *arguments*.

The head and body of a clause are all examples of *terms*. In general, a term is a *variable*, an atom, or a cons of any two terms.

A variable is written as a symbol beginning with a single character '?'. An *anonymous variable*, i.e. a variable having just a single occurrence in the clause, may be written as the single character '?'. In the implementation, variables are represented as locatives. Occurrences of variables inside structures are represented as *invisible pointers*, described in more detail in Section 5.3.

## 2.4 Declarative and procedural semantics

The declarative semantics of an LM-Prolog statement is no different from that of other Prologs and more generally from that of predicate logic. It defines the set of predications which are true according to a given program. With Warren [1977] we say that a predication is *true* if it is the head of some clause instance and each of the predications of that clause instance is true, where an *instance* of a clause is obtained by substituting, for some of its variables, a new term for each occurrence of the variable.

Not all true predications are heads of clause instances, however, the exceptions being *built-in predicates* (called *evaluable predicates* in other references), the declarative semantics of which will have to be accounted for separately. The principal built-in predicates are listed in Appendix B.

The procedural semantics of an LM-Prolog statement is a superset of that of other Prologs since LM-Prolog has a richer control language. Given a predication $P$, the system searches the clauses of a definition of $P$, in the same order as they were defined, for a matching head. The search is restricted to the current *universe* which is a sequence of worlds. The systems searches the worlds of the universe, one at a time, for a definition of $P$. The search normally stops at the first definition found. If no definition is found, an error is signalled.

If a matching clause instance is found, each of the predications in the body of the matching clause instance is executed, from 'left' to 'right'. If at any time a match is not found, the system backtracks. Execution terminates with success if there are no more pending predications. Execution terminates with failure if all choices for the original predication have been rejected and there are no more choices.

Predicates can be declared to be mutable, allowing programs to update the database dynamically. The universe can similarly be updated dynamically by adding or deleting worlds.

A predicates is deterministic if it can have at most one solution whatever the arguments. The control language of LM-Prolog includes the ability to declare a predicate deterministic, i.e. to force it to never produce more than one solution.

## 2.5 Programming comments

The parser issues stylistic warnings for variables other than ? if they occur only once. This catches many typing errors. No warnings are issued for variables beginning with ?ignore.

Unlike most Prologs, *undefined predicates* signal errors when called, since we believe such behavior to be more informative than just failing. Note that it is possible to define empty predicates, e.g. fail can be defined in LM-Prolog. Calling a compiled predicate with the *wrong number of arguments* also signals an error.

Predicates in LM-Prolog can take any number of arguments—they do not have fixed *arity*. This might seem like a deviation from first-order language theory but it is not. LM-Prolog predicates can be viewed as taking exactly one argument which is a list.

# 3. Overview of LM-Prolog implementation

## 3.1 Introduction

Prolog is a language where procedure calls and unification are by far the most common operations, hence it is important to minimize the overhead incurred by these. With this in mind, the crucial issues in the implementation of LM-Prolog are the following, to be discussed in more detail in subsequent sections.

1. Organization of the database.
   The concept of worlds makes the structure of the database non-trivial. The most important problem becomes to minimize the overhead involved with searching the sequence of worlds. Indexing techniques for speeding the search for matching clause instances are also relevant.

2. Organization of run-time data structures.
   The main design decision is how to represent terms at run time, and the choice between "structure sharing" and "copying pure code". We have chosen a variant of copying pure code because (*i*) the run time terms become indistinguishable from the corresponding Lisp objects, (*ii*) the firmware provides good support for constructing terms whereas the machine architecture does not provide support for handling structure sharing 'molecules', i.e. pairs ⟨*source term, frame*⟩.

3. Unification.
   Two problems have to be solved viz. (*i*) the unification of two object terms, and (*ii*) the unification of a object term with a source term. The latter problem is aided by a coding scheme for source terms enabling us to use dispatch hardware to support unification. Our unifier can operate in three different modes with respect to cyclic terms, viz. (*i*) ignoring the possibility of cyclic terms being created and related termination problems, (*ii*) preventing cyclic terms from ever being created using a traditional occur check, and (*iii*) handling cyclic terms i.e. being able to unify, copy, and print them, and pass them to Lisp.

4. Run-time control structure.
   We use a continuation-passing control structure as outlined in Chapter 1. Each predicate compiles to a Lisp function that takes an extra argument which is a continuation that in the general, non-deterministic case corresponds to the remaining goals to be solved. If the predicate succeeds it calls the continuation, if it fails it returns nil. This control structure is optimized if a compile-time analysis deems a predicate deterministic.

## 3.2 Database organization

Conceptually, the database is organized as a collection of worlds, each world being a collection of predicates. At any given time, the current universe imposes a search order over the database. Two different predicates with the same name may coexist in different worlds. The :if-another-definition declaration (see define-predicate) decides what the semantics for that case is.

The implementation maintains the current universe in the special variable *universe*.

Each predicate is stored as a data structure in the database. This data structure contains a function called the *prover*, which is invoked by predicate-to-predicate calls. The kind of prover depends on the kind of predicate:

- For indexed predicates, the prover tries to find an index to apply. A subterm of one of the arguments passed in a call is used as a lookup key in the index, and typically a small set of clauses are found and tried sequentially.
- For static compiled un-indexed predicates, the prover is the translation of the entire predicate.
- Otherwise, the prover is a small function that tries the clauses sequentially.

For dynamic compiled predicates and for indexed compiled predicates, each clause is compiled to a function. Otherwise, the clauses are just stored as source terms.

Provers of compiled predicates are exemplified in Appendix C. A typical prover of an interpreted predicate a could look like the following:

```
(defun a-in-user-prover (continuation &rest arguments)
  (rest-arg-fixup arguments)
  (try-each-interpreted-clause
    (contents a-in-user-clauses) continuation arguments *trail*))
```

An interpreted predicate a that is forced to be deterministic might look as follows:

```
(defun a-in-user-prover (continuation &rest arguments)
  (rest-arg-fixup arguments)
  (cond ((try-each-interpreted-clause
          (contents a-in-user-clauses) (continuation (true)) arguments *trail*)
         (invoke continuation))))
```

Above, contents denotes the operation of following a locative pointer. The function rest-arg-fixup fixes a problem with &rest arguments. The special variable *trail* contains the trail stack pointer and is discussed in section 5.5. The special form continuation creates a continuation of its argument.

## 3.3 Term organization

The LM-Prolog implementation deals with two kinds of terms, *object terms* (or, simply, *terms*, to be discussed in the following section), and *source terms* which are descriptions of object terms. The source terms contain various codes that the microcoded unification routines dispatch on. There are special codes for single, first, and subsequent variable occurrences, since the unification routines take different actions for different kinds of occurrences. The codes are defined in Section 5.1.

Object terms are terms ordinarily dealt with at run time and are visible to the user, whereas source terms occur in the database and as constants inside of compiled code. Source terms are used as templates for creating object terms. The creation of object terms is expensive in terms of both space and time, and in all non-structure sharing implementations it is important to not wastefully create terms. Indeed, LM-Prolog creates object terms only when a variable is matched with a source term and prior to predicate-to-predicate calls.

For example, the source term corresponding to the clause

```
((append (?a . ?x) ?y (?a . ?z)) (append ?x ?y ?z))
```

is

```
(3 (3 (0 . append) 3 (3 (1 0 . ?a) 1 1 . ?x) 3 (1 2 . ?y) 3 (3 (2 0 . ?a) 1 3 . ?z) 0)
     3 (3 (0 . append) 3 (2 1 . ?x) 3 (2 2 . ?y) 3 (2 3 . ?z) 0) 0)
```

If one may attempt an analogy with micro-biology, source terms are the genes for object terms, and unification is the ribosome, synthesizing object terms only when needed.

## 3.4 Unification

Since there are two different kinds of terms, there are also two different unification algorithms in the system. The most commonly used algorithm, $Unify_S$, is the one that unifies a predication (represented as a object term) with the head of a clause (represented as a source term). This is the main action taken by the interpreter. Compiled code does not construct predications, but rather their arguments, and unifies these with corresponding source terms for arguments of the head of a clause, one at a time. The second algorithm, $Unify_O$, which unifies two object terms, is used as a subroutine to $Unify_S$. So is a third algorithm, *Construct*, that constructs object terms out of source terms. These three algorithms correspond to three instructions of the extended instruction set, see Appendix D.

## 3.5 Invocation of predicates

Invocation of predicates rests upon the standard function calling protocol of the machine which corresponds to call-by-value in conventional languages. Even a logical variable is passed by value, in a sense, since it is the pointer to its value cell that is passed.

There are two complications to this that deserve further comment. One has to do with the dynamic search order of the database, the other is non-determinacy.

### 3.5.1 The cache mechanism

When a predicate $P$ is called, the system in general has to search for its definition, according to the current universe. To avoid having to do this search if the universe is not changing, we have invented a cache mechanism. The definitions of $P$, if any, are stored under prolog-definitions on $P$'s property list. The value of the property is an *alist* (association list)

*(cache (world . definition) (world . definition) ...)*

where the first element is a distinguished element

*(universe . current-definition)*

The cache mechanism consists in checking if *universe* is identical to *universe* (the current universe). If so, use *current-definition.* If not, the rest of the alist is searched according to *universe*, and *cache* is set up. There is a dedicated machine instruction for predicate-to-predicate calls which handles the normal case of the cache mechanism.

### 3.5.2 Backtracking

Variable bindings generally have to be undone upon backtracking. When a value cell is bound, its address is also pushed onto the *trail-stack*. The value cell is eventually made unbound when the trail-stack *unwinds*.

Backtracking is implemented by (*i*) for each backtrack-point saving away the trail-stack pointer; (*ii*) continuation passing; (*iii*) if the value passed back from the continuation is nil, unwinding the trail-stack and reinstalling its stack pointer, and continuing to find more solutions.

Assume that the interpreter is called with a continuation $C$ and a conjunction of two predicates $P$ and $Q$. The interpreter will then call $P$ with continuation $C_1$

$$C_1 = [\text{call } Q \text{ with continuation } C]$$

If $P$ succeeds it will invoke its continuation $C_1$ whereby $Q$ is called. $P$ will go on to consider more alternatives only if $C_1$ returns nil. Thus, the way to force a predicate to exhaust its search space is by calling it with continuation false. To get only the first solution, one calls it with continuation true. The top-level calls the interpreter with continuation to ask the user whether more solutions are wanted. Sets and bags use special continuations that collect the answers, and so on.

## 3.6 Non-determinacy and continuations.

Continuations are normally passed as arguments in predicate-to-predicate calls. The only exception to this is when a continuation is put on the trail as explained in Section 5.3.

In the current version of the Lisp system, continuations are allocated as lists where the first element is a function $F$ of the rest of the elements which are the variables referred to by $F$. In other words, all local variables that the continuation needs to access are copied when the continuation is created. A continuation is *invoked* by applying its car to its cdr. $F$ will then get all its variables passed as arguments. We have adopted this solution since compiled code can only access (*i*) the local activation frame and (*ii*) *special* i.e. dynamically scoped variables which are too expensive for our purposes.

In newer versions of the Lisp system, *lexical scoping* of variables will be supported. With lexical scoping, "internal" functions, for example continuations, will have access to the activation frame of the function they occur in. For the normal use of continuations, then, there will be no need to allocate continuations as lists. Instead we will really be using functions of zero arguments as continuations. A continuation will be invoked by calling it.

For example, the Lisp procedure f defined as

```
(defun f (x y)
  (g x (continuation (h y))))
```

passes a continuation to the procedure g. The continuation refers to a variable in the stack frame of the invocation of f. Without lexical scoping, (continuation (h y)) would macro-expand to (list #'h y) i.e. at run time, a list 2 long would be constructed. With lexical scoping, the continuation form would macro-expand to #'(lambda () (h y)) i.e. at run time, a pointer to a piece of compiled code referring to a higher binding context would be passed. Nothing would be constructed.

Using continuations can require a large execution stack since in general the system recurses deeper while it succeeds and returns only upon failure. Every node in the proof tree will correspond to a stack frame. Clearly this is unacceptable, and so reasonable Prolog implementations optimize "deterministic subtrees". LM-Prolog does a compile time analysis of predicates to see if they are deterministic. Predicates can also be forced to become deterministic by a declaration (see define-predicate). The compile time analysis is based on detection of incompatible clause heads and on knowledge about the predicates called in the bodies of clauses.

Assume, again, that the interpreter is called with a continuation $C$ and a conjunction of two predications $P$ and $Q$. Assume moreover that $P$ is known, by compile-time analysis, to be deterministic. The interpreter will then call $P$ with continuation true. If $P$ succeeds it will invoke its continuation true and return t. The interpreter then goes on to call $Q$ with continuation $C$. The stack space used in the computation of $P$, and all sub-computations, is thus reclaimed and reused in the computation of $Q$. No intermediate continuation is created.

For compiled code, analogous code is emitted in the non-deterministic and deterministic cases.

## 3.7 Tail recursion

When a function $f$ finishes its computation by calling another function $g$ we say that $f$ calls $g$ *tail-recursively*. If $f = g$, we say that $f$ is *locally tail-recursive*. Tail-recursive calls offer an important opportunity for optimization, particularly in applicative languages. On the Lisp machine, a tail-recursive call makes it possible to reclaim the stack frame of the calling function. Indeed, the machine can be told to operate in a mode in which it always reclaims the stack frame at tail-recursive calls.

In Prolog, tail-recursion occurs when a predicate $p$ calls another predicate $q$ as its last goal, and $p$ has no more alternatives. Some implementations of Prolog optimize tail-recursive calls, notably [Warren, 1980] and [Bruynooghe, 1982a]. LM-Prolog's interpreter detects opportunities for tail-recursion optimization, and is aided in this respect by determinacy declarations. The compiler transforms local tail-recursion into iteration by post-processing the emitted Lisp code by a recursion removal package based on [Risch, 1973].

## 3.8 The interpreter

The interpreter is tiny given unification and given that all built-in predicates have their own provers. Its main functions are prove-conjunction and try-each-clause which handle conjunctions of predications and alternative clauses, respectively. It is small enough to be listed verbatim in Appendix A. Aspects of the interpreter are discussed in Chapter 6.

## 3.9 The compiler

The compiler is written in Lisp and emits Lisp code, which is post-processed as described above. The emitted code makes use of the extended instruction set described in Appendix

D if microcode support is available. It is further translated into machine code by the ordinary Lisp compiler.

The compiler operates in one mode for static un-indexed predicates and in another mode for dynamic or indexed predicates. In the previous case it translates predicates to prover functions. In the latter case, it translates individual clauses to functions.

The compiler optimizes static predicates in the following ways:

Arity restriction.
: In general, an LM-Prolog predicate does not have a fixed arity and must be prepared to handle predications of any arity. For static predicates, all clause heads are known and so the compiled code can be specialized to handle only the relevant arities.

Incompatible head detection.
: Even for un-indexed predicates there are opportunities to not just try the clauses sequentially. Using '+'-declarations of arguments (see define-predicate), the compiler can detect mutual exclusion among the clause heads.

Determinacy analysis.
: For a static predicate, the compiler performs an analysis to see if it is deterministic. The principal use of this analysis is to optimize subsequent compilation of predications of this predicate. Incompatible head detection often helps this analysis step.

The main parts of the compilation process can be divided into:

- The optimizations discussed above.
- Compilation of alternative clauses at the top-level of a predicate and when opencoding (unfolding) a predication. This involves the resolution of predications against clause heads producing instances of clause bodies unification and variable initialization goals.
- Compilation of conjunctions of predications. This involves continuation passing and predicate-to-predicate calls.
- Compilation of unification.
- Compilation of built-in predicates which is handled by *intrinsic compile-methods.*

Minor optimizations are performed for all these steps.

## 3.10 Lisp interface

We have shown how the Lisp-to-Prolog interface works by the continuation passing protocol. As a variant of that control structure, the relation can be computed in a *stack group* i.e. in a coroutine object, with continuation to detach (stack-group-return) from the coroutine with its state saved. Resuming the coroutine triggers backtracking. This control structure corresponds to using a Prolog relation as a generator of solutions. On top of this mechanism, we have defined the flavor prolog-query with methods

:flush Kill the generator and release all its resources.

:next-answer

Request the next answer. An instantiation of the answer is returned.

The default top-level behavior is based upon prolog-query.

There is also a Prolog-to-Lisp interface, defined by built-in predicates (see lisp-value, lisp-predicate, and lisp-command). The compiler does not translate calls to these predicates to calls to the Lisp evaluator, but rather tries to compile their arguments as far as possible. Normal object terms are compatible with (indeed, indistinguishable from) Lisp objects, and so the arguments of these predicates are just passed to the called Lisp procedures. In case the Lisp procedure wants to do something strange to its argument, like storing it away, or in case certain language extensions have given arise to flavor instances among the Prolog terms, the arguments have to be converted first. The interface predicates take an optional control annotation which notifies LM-Prolog that conversion is needed.

# 4. Database organization

As we have discussed in sections 2.3 and 3.5, LM-Prolog has facilities to support worlds i.e. collections of predicates, and user controllable indexing.

LM-Prolog's database is a collection of definitions, subdivided into worlds. A world is identified by a symbol. Not all worlds need to be *active*. A definition is active if it belongs to a world that is in the universe. When a predication $P$ is to be executed, the system in principle searches the worlds of the universe, one at a time, for a definition of $P$. The search stops at the first definition found, unless a different semantics is desired by the :if-another-definition option (see define-predicate).

Each predicate is stored in the database as a data structure with the following slots:

- prover—a function of the continuation and the predication's arguments.
  This is the entrypoint to the code that executes the predicate. It can be compiled code for the predicate, or a small function which serves as an interface to the interpreter. The calling sequence is standardized to be the same for both cases.
- interpreter-slot—is a substructure containing:
  - deterministic—non-nil if the predicate is deterministic.
    The compiler analyzes all static compiled predicates to see if they are deterministic. The user may also optionally declare a predicate to be deterministic, see define-predicate.
  - interpreter-prover—the prover or nil. Under certain circumstances, interpreter-to-predicate calls can be optimized to not go through the prover, but rather proceed directly to try the clauses of the callee. The value of this slot is equal to the disjunction of the following conditions, i.e. the optimization is performed if all the conditions are false:
    - The callee is indexed.
      It is normally worth while to go through the index.
    - The callee is compiled.
    - The callee contains cut.
      The callee's prover must take special actions for the workings of cut, see Chapter 6.
    - The value of the :if-another-definition option is not :ignore (see define-predicate).
  - clauses—the clauses making up the definition.
    The structure of a clause is discussed below.
- indexer—function for adding new clauses
- predicator—name of predicate defined
- options—given and compiler-emitted options merged with the defaults
- indexes—mainly hash tables for indexes of predicate

Clauses are flavor instances with instance variables

**templates**
A cons of source term for the head and source term for the.body. Source terms are explained in Section 5.1.

**committing**
t iff the head of the clause is incompatible with the heads of all subsequent clauses. The value of this slot is computed by the Incompatible Head Detection optimization step discussed in Section 7.2.

**predicator**
name of predicate defined.

**name** the name of a named clause, or "No name" for an unnamed clause.

**contains-cut**
t iff the clause contains a cut.

**prover** a function of a continuation and arguments. Used for compiled indexed and compiled dynamic predicates.

**world** the world that this is a part of

Interesting flavor methods of clauses are:

**:unify** *term*
unifies *term* with a unique instance of the clause, this is used by some database predicates.

**:resolve** *continuation arguments*
just calls the prover, this is used for compiled indexed and compiled dynamic predicates.

# 5. Term organization

## 5.1 Representation of source terms

A source terms $S$ is represented as a cons whose car contains a *type code* (an integer) with the following meaning:

0. $S$ stands for a ground term. The cdr contains the ground term.
1. $S$ stands for a first occurrence of a variable. The cdr contains an *occurrence spec* as explained below.
2. $S$ stands for a subsequent occurrence of a variable. The cdr contains an occurrence spec.
3. $S$ stands for a non-ground cons. The cdr is a cons of {source term for the car} and {source term for the cdr}.
4. $S$ stands for a read-only first occurrence of a variable. The cdr contains an occurrence spec.
5. $S$ stands for a read-only subsequent occurrence of a variable. The cdr contains an occurrence spec.
6. $S$ stands for a read-only single occurrence of a variable. The cdr contains the variable name or nil.
7. $S$ stands for a single occurrence of a variable. The cdr contains the variable name or nil.

Read-only annotations are explained in Chapter 8. Read-only single occurrences are probably not useful but are included for symmetry.

An *occurrence spec* is either a cons (*occurrence code . variable name*) or just *occurrence code.* The latter is an integer $xyy$ where $x$ (octal) specifies a base register and $yy$ (octal) is an offset. The following base registers are used:

0. The interpreter's scratch vector.
1. A compiled predicate's local variable block.
2. A compiled predicate's argument block.

Base register 0 occurs in clauses in the database. The interpreter and the compiler use those. Compiled Prolog code uses base registers 1 and 2.

## 5.2 Representation of object terms

This section discusses the representation of run-time (object) terms. The discussion takes place from two viewpoints. First, we describe the representation from the viewpoint of Prolog, of the user, and of normal Lisp machine functions. In order to be able to describe the implementation of logical variables, however, we must also describe these matters from the viewpoint of the firmware and hardware.

### 5.2.1 The normal view

From the normal point of view, a Prolog term is indistinguishable from the corresponding Lisp object, e.g. 3.14 is both a Lisp and a Prolog flonum; () is the empty list in both Lisp and Prolog; (lm . prolog) is a cons of two symbols in both languages.

The one exception of the above statement is the treatment of logical variables. An (unbound) logical variable really doesn't correspond to a Lisp object at all since it stands for an unknown term. Nevertheless, it has to be represented somehow. Lisp machine functions see (unbound) variables as locatives. Moreover, if a term $T_0$ contains one or more occurrences of an unbound variable $V$, and $V$ becomes bound to a term $T_1$, then $T_1$ **effectively** substitutes all occurrences of $V$ in $T_0$. The implementation of course doesn't use substitution, but from normal viewpoints, the result is indistinguishable from that of substitution.

The next subsection discusses the mechanisms that realize this.

### 5.2.2 A machine-oriented view

To understand the machine's point of view, we must first discuss its storage conventions. On the firmware level, a term is a tagged pointer $\langle datatype, pointer\rangle$. For the sake of this discussion we will use symbolic names prefixed by dtp- to denote datatypes. The datatype field determines the kind of term:

- variables are dtp-locative,
- conses are dtp-list,
- other datatypes are atoms.

For the sake of storage efficiency of list structures, *machine words* have a third field called the *cdr-code*. For the sake of this discussion we will use symbolic names prefixed by cdr- to denote cdr-codes. The cdr-codes are used as follows:

- A cons $\langle dtpa, ptra\rangle.\emptyset$ is stored as
  $p$: $\boxed{\langle \text{cdr-nil}, dtpa, ptra\rangle}$
- A cons $\langle dtpa, ptra\rangle.u$ where $u$ is a cons is stored as
  $p$: $\boxed{\langle \text{cdr-next}, dtpa, ptra\rangle}\boxed{\text{storage for } u\ldots}$
- A cons $\langle dtpa, ptra\rangle.\langle dtpd, ptrd\rangle$ where $\langle dtpd, ptrd\rangle$ is not a cons or $\emptyset$ is stored as
  $p$: $\boxed{\langle \text{cdr-normal}, dtpa, ptra\rangle}\boxed{\langle —, dtpd, ptrd\rangle}$

where $p$: $\boxed{\langle\ldots\rangle}\boxed{\langle\ldots\rangle}$ denotes adjacent machine stored at address $p$. In all three cases, the cons term itself is represented as a tagged pointer $\langle$dtp-list, $p\rangle$.

Thus, it takes one word per element to represent a list (not including the space requirements of the elements themselves). Accessing the car of a cons $\langle$dtp-list, $p\rangle$ consists in reading the machine word stored as $p$, ignoring the *cdr-code* field. Accessing the cdr of a cons $\langle$dtp-list, $p\rangle$ consists in reading the machine word stored as $p$, and then taking further action depending on the *cdr-code* field.

Now, there is one important exception to the storage scheme outlined here. If the datatype field of a machine word contains an *invisible pointer* datatype such as dtp-external-value-cell-pointer, then the pointer field is considered as a forwarding pointer. For example, the car of ⟨dtp-list, *p*⟩ with

*p*: | ⟨cdr-nil, dtp-external-value-cell-pointer, •⟩ | . . . | ⟨—, *dtpa*, *ptra*⟩ |

is ⟨*dtpa*,*ptra*⟩. Invisible pointers can form chains of any length. The car and cdr operations follow such chains to their end. Invisible pointers are used for representing occurrences of variables inside structures, and also for representing variables linked to other variables. The invisible pointer chains thus created always end inside a *value cell.* Value cells are the subject of the following subsection.

**5.2.3 Value cells and binding**

A logical variable is represented as a tagged pointer ⟨dtp-locative, *p*⟩, pointing at a value cell. A value cell contains a pointer to a term. In the unbound case, a value cell points to itself. For the sake of a more friendly appearance to the user, a value cell may optionally contain the source code variable name. Value cells of the interpreter contain the source code variable name, those of compiled code do not. Thus, a value cell is stored as one machine word

*p*: | ⟨cdr-nil, dtp-locative, •⟩ |

or as two machine words

*p*: | ⟨cdr-normal, dtp-locative, •⟩ | ⟨—, dtp-symbol, •⟩ | . . .

. . . | storage for source code name of the variable |

A value cell can become bound to a term in which case the *datatype* and *pointer* field of the first machine word are replaced by the term's. Again, there is an exception if the term is a variable, in which case the datatype stored is changed to dtp-external-value-cell-pointer. We can have chains of variables bound to other variables by forwarding pointers. A chain ends either in an unbound variable or in a term which is not a variable. The following of chains of pointers is called *dereferencing* and is performed automatically when a car or cdr indirects to a value cell. Dereferencing has to be done explicitly on terms stored as local variables in compiled predicates. Unbound variables are self-referential so that the unifier can get hold of an unbound variable after dereferencing it.

It is the usage of forwarding pointers to value cells inside of the representation of terms that give the **appearance** of literal substitution of occurrences of a variable when that variable becomes bound.

As opposed to e.g. DEC-10 Prolog [Warren, 1977], there is no enforcement of the direction of the pointer when a variable is bound to another variable. That is to say, the *senior variable* can be assigned to the value cell of the *junior variable*, equally well as the other way around. It is unclear how an enforcement could be implemented, due to the complicated storage allocation scheme of the machine (see Section 5.5). In DEC-10 Prolog, the enforcement is cheap (an integer comparison of two registers) and necessary to prevent a dangling pointer problem. Warren claims that the enforcement also helps preventing long chains of linked variables to be built up.

### 5.2.4 An example

As an example of a complex term, let us once again consider the clause

((append (?a . ?x) ?y (?a . ?z)) (append ?x ?y ?z))

which is represented as

$\boxed{\otimes}\boxed{\otimes}\boxed{\otimes}\boxed{\otimes}$ ((append (● . ●) ● (● . ●)) (append ● ● ●))

where $\boxed{\otimes}$ denotes an unbound value cell and ● denotes a variable occurrence.

### 5.2.5 &REST arguments

As we have seen, variable occurrences inside of list structures are always represented as invisible pointers to value cells. However, lists can be implicitly constructed as &rest arguments by the predicate calling mechanism, when the callee accepts a variable number of arguments. If elements of a &rest argument are variables, the mechanism will not automatically change the datatype to dtp-external-value-cell-pointer. For this case, the implementation uses a runtime function, rest-arg-fixup for the datatype correction.

Another problem with variable arity is the fact that &rest arguments reside on the execution stack and not in heap storage

### 5.2.6 Some notes on complexity

The cost of making a variable binding is constant.

The cost of accessing a value cell is linear in the length of the chain of forwarding pointers starting at the value cell, although the proportional constant is quite small (6 $\mu$-cycles).

The cost of constructing an object term out of a source term is roughly proportional to its number of occurrences of type codes 1, 3, 4, 6, and 7.

## 5.3 Unification

### 5.3.1 Introduction

Unification [Robinson, 1965] is the heart of all Prolog implementations, indeed, of all resolution-based logic programming. It plays the rôle of constructor and accessor functions in functional programming languages, and of conditionals, assignment, and pointer referencing in conventional languages.

In Robinson's *classical* unification algorithm, the domain of unifiable terms is restricted to acyclic terms. Contemporary implementations of Prolog typically do not enforce the acyclicity of terms and it is easy to construct cases where unification does not terminate. Lately, several authors have proposed *unrestricted* unification algorithms which consider a variable to be unifiable with a term containing it. See e.g. [Huet, 1976; Robinson, 1976; Colmerauer, 1980; Haridi, 1981].

With respect to the treatment of cyclic structures, our system operates in one of three *modes*:

*:ignore* This is the naïve way of treating cyclic terms: the possibility of constructing them is ignored altogether, and attempts to unify such terms may not terminate. This mode of operations is the fastest (almost linear time) and in most Prologs the only available mode.

*:prevent* This is the classical way of treating cyclic terms: the possibility of constructing them is prevented by means of an *occur check* which is invoked any time that a variable is bound to a cons term. The occur check simply does a tree walk of the term, searching for an occurrence of the variable. This operation mode slows down unification severely, making its time complexity $O(n^2)$ in the size of the terms. It should be noted that there exist better (indeed, $O(n)$) algorithms, although for small $n$ they are much worse than the algorithm we use. See [Martelli and Montanari, 1976] and [Paterson and Wegman, 1976] for these linear algorithms.

*:handle* The possibility of constructing cyclic terms is fully supported by unification and by the rest of the run-time system. In this mode of operation, unification still proceeds in almost linear time, although with a larger constant. It is well known that cyclic terms may introduce paradoxes and jeopardize the consistency of the logic system, but since cyclic terms may have pragmatic advantages, they are included as an optional feature. The semantics of logic programming languages with cyclic terms is a difficult subject, treated first in [Colmerauer, 1980] where, in fact, all connections with logic have been severed. Later, van Emden and Lloyd [1984] have shown that Colmerauer's language can be regarded as a logic programming language. Hansson *et al.* [1981] attack the problem by distinguishing between canonical (i.e. finite) and noncanonical terms and having different computation rules for the two cases.

### 5.3.2 Classical unification of object terms

The classical unification algorithm is very simple. Our formulation closely matches e.g. the formulation of *unify* given in [Morris, 1980].

1. Two identical terms unify. We use the Lisp predicate eq to judge whether terms are identical.
2. Two cons terms unify if their cars unify and their cdrs unify.
3. A variable unifies with another term if it *binds* to the other term. In *:prevent* mode, a variable binds to a term iff the term does not contain the variable. In the other modes, a variable always binds to a term.

4. Two equal atomic terms unify. We use the Lisp predicate equal to judge whether atoms are equal.
5. No other pairs of terms unify.

As we shall see later in this section, there is a way for objects of user-defined datatypes to unify with other objects.

As an example of unifying object terms, let us consider the predication

```
(= (?a ?b 0) (?b ?c ?a))
```

with all variables initially unbound. It is represented as

$\boxed{\otimes}\,\boxed{\otimes}\,\boxed{\otimes}$ (= (• • 0) (• • •))

Satisfying the predication results in the data structure

$\boxed{\bullet}\,\boxed{\bullet}\,\boxed{0}$ (= (• • 0) (• • •))

which prints as

```
(= (0 0 0) (0 0 0))
```

### 5.3.3 Unrestricted unification of object terms

In *:ignore* mode, the classical algorithm may not terminate if its arguments are cyclic. To arrive at unrestricted unification with guaranteed termination, we modify the algorithm slightly introducing assumptions about identity, replacing steps 1-2 above by:

1'. Two terms unify if they are identical or if they are assumed to be identical.

2'. Two cons terms $\alpha.\beta$ and $\gamma.\delta$ unify if, under the assumption that $\alpha.\beta$ and $\gamma.\delta$ are identical, $\alpha$ unifies with $\gamma$ and $\beta$ unifies with $\delta$.

Our idea is to use these assumptions to narrow down the structures to be unified until a base case can be used. Variants of the scheme have been published in [Huet, 1976; Robinson, 1976]. Interesting alternative schemes based on equations can be found in [Colmerauer, 1980; Haridi, 1981].

Morris has shown [1980] our idea to be equivalent (*i*) to an algorithm by Fisher and Galler [1964] for representing an equivalence relation in such a way that it may expeditiously be both interrogated and progressively coarsened, (*ii*) to an extension of the classical unification algorithm so that it computes a substitution which contains cons-cons pairs in addition to variable-term pairs.

Since the size in cons cells of a cyclic term is finite and each assumption decreases the effective total size by one, a formal termination proof based on induction on the effective total size can be worked out and has been published in [Fages, 1983].

The time complexity of the algorithm depends crucially on how fast the assumptions can be checked. It is important that the assumptions are dereferenced, [1] i.e. that chains of identity assumptions are followed to the end, when new assumptions are added. Dereferencing assumptions yields trees of terms that are assumed to be unifiable. The algorithm has been shown to be quadratic if the assumptions are not dereferenced, $O(n \log n)$ if they are dereferenced, and almost linear if, whenever an assumption is added, it is added from a smaller assumption tree to a larger. [2] See [Galler and Fisher, 1964; Tarjan, 1975] for details.

Our implementation dereferences assumptions but is ignorant about the sizes of the assumption trees. However, checking the assumptions is extremely fast and uses the same mechanism as dereferencing, and so we claim that we have an almost linear algorithm. In particular, when the algorithm operates on strict trees, assumptions are made but are never encountered. We use the Lisp machine primitive bind to temporarily insert forwarding pointers into the terms to be unified. If $\alpha.\beta$ stored at address $p$ is assumed to be identical to $\gamma.\delta$ stored at address $q$, then the situation before and after unification can be described as:

$p$: $\boxed{\alpha}$ … $\boxed{\text{storage for } \beta \ldots}$
$q$: $\boxed{\gamma}$ … $\boxed{\text{storage for } \delta \ldots}$

With the identity assumption in effect, the situation is:

$p$: $\boxed{\langle -, \text{dtp-header-forward}, \bullet \rangle}$ … $\boxed{\text{storage for } \beta \ldots}$

$q$: $\boxed{\gamma}$ … $\boxed{\text{storage for } \delta \ldots}$

with $\boxed{\alpha}$ to be reinstalled at $p$ when the special binding stack unwinds. The difference between dtp-external-value-cell-pointer and dtp-header-forward is that the former forwards a car or a cdr, whereas the latter forwards a cons.

### 5.3.4 Variable Binding

The variable binding mechanism has been discussed in Section 5.2.

In order for the implementation to be able to backtrack, it must in particular be able to undo variable bindings. The traditional way in Prolog implementations to arrange this is using a "trail", i.e. a stack onto which the system pushes pointers to value cells being bound. LM-Prolog is no different in this respect. The following comments are relevant though.

- LM-Prolog trails **all** variable bindings since there is no easy way for it to see if the trailee would in any case be discarded on backtracking since there is no machine register holding the "latest backtrackpoint".

[1] Called *path compression* in [Tarjan, 1975]
[2] Called *balancing* in [Tarjan, 1975]

- LM-Prolog uses separate trails in the implementation of lazy and eager collections. Each LM-Prolog toplevel has its own trail.
- Continuations are allowed on the trail and are invoked when the trail unwinds. This is used when there is a need to guarantee that something happens on backtracking, e.g. to undo a side-effect.

If the occur check is enabled, the variable binding mechanism traverses the other term and fails if the other term has an occurrence of the variable about to be bound. In general, this makes unification quadratic in the size of the largest of the two terms.

### 5.3.5 Unifying object terms with source terms

Recall that a source term is a cons whose car is a type code:

(0 . *"ground constant"*)
(1 . *"offset for first variable occurrence"*)
(2 . *"offset for subsequent variable occurrence"*)
(3 *"car's source term"* . *"cdr's source term"*)
(4 . *"offset for read-only first variable occurrence"*)
(5 . *"offset for read-only subsequent variable occurrence"*)
(6) *;;read-only single variable occurrence*
(7) *;;single variable occurrence*

The algorithm follows (numbers corresponding to type codes). *OT* denotes the object term.

0. $Unify_O$ *OT* with *"ground constant"*.
1. Store *OT* in the place located by *"offset for first variable occurrence"*.
2. $Unify_O$ *OT* with contents of place located by *"offset for subsequent variable occurrence"*.
3. If *OT* is a variable, then bind it to the object term that this source term describes. If *OT* is a cons, then $Unify_S$ its car with *"car's source term"* and its cdr with *"cdr's source term"*. Otherwise fail.
4. If *OT* is a variable, then allocate a new cell *X* and store it at the proper offset and bind *OT* to a *read-only object* (see Chapter 8) made of *X*.
5. Retrieve *X* at the proper offset. If *OT* and *X* are variables, make a read-only object of *X* and bind *OT* to it. Otherwise if *X* is a non-variable, then $Unify_O$ *OT* with *X*.
6. If *OT* is a variable, then bind it to a read-only object.
7. Succeed.

The issues having to do with cyclic structures do not come up here (except in cases 2 and 5), since source terms cannot be cyclic in our implementation.

Note that the first occurrence of a variable in a clause corresponds to its own type code. Many variables are initialized by their unification with some term. The remaining variables have to be initialized by allocating value cells. We haven't done any statistics on this but it seems that more than 50% of the variables in typical programs are initialized by unification.

Note that there is just one case where a term is actually constructed: when a variable is unified with a "type 3" source term. The constructing process is straightforward and its output is as follows. Numbers correspond to the type code of its input.

0. *"Ground constant"*.
1. A newly allocated value cell, which is also stored in the place located by *"offset for first variable occurrence"*.
2. The contents of the place identified by *"offset for subsequent variable occurrence"*.
3. A cons of the construction of *"car's source term"* and the construction of *"cdr's source term"*.
4. A read-only object made out of a new value cell, which is also stored in the place located by *"offset for first variable occurrence"*.
5. The contents of the place identified by *"offset for subsequent variable occurrence"* is retrieved. It is returned if bound, otherwise a read-only object made out of it.
6. A read-only object made out of a new value cell.
7. A new value cell.

The cons constructed in case 3 follows the conventions described in Section 5.2, namely, if its car or cdr is a variable, then the pointer physically stored in the cons cell will be an invisible (= forwarding) pointer to the variable's value cell.

The construction process also uses cdr-coding to compactify lists as much as possible. It does not really allocate cons cells but rather chunks of memory to make cdr-coded lists.

#### 5.3.6 User extensions

The system is designed to allow modular definitions of new data types as flavors. Adding new data types to Prolog is made possible by the fact that the algorithm for unifying two object terms has one additional step:

> A flavor instance can be unified with a non-variable if the instance accepts a :unify message with the non-variable as argument. The unification fails iff the flavor's :unify method returns nil.

The restriction to non-variables is because variables already unify with any terms, including flavor instances.

So in order to add a data type, the user needs to define a flavor (which should be a sub-flavor of prolog-flavor) and define its :unify method.

Lazy collections, eager collections, and constraints are examples of system extensions that were implemented this way. These extensions and their :unify methods can be found in Chapter 8.

## 5.4 Instruction set extensions

By writing a moderate amount of microcode by hand, we have introduced a number of instructions to the benefit of the implementation. The extensions have led to a factor of 3 speed up of typical compiled predicates and about a factor of 2 more compact code for compiled predicates. The difference in compactness of code is due partly to the fact that some of the new instructions replace opencoded (unfolded) function calls, partly because the microcode support makes the *Unify*$_S$ algorithm feasible in compiled predicates.

The CADR micro-processor [Knight *et al.*, 1981] has an instruction set which is remotely reminiscent of that of the PDP11 computer series. Micro-programming on the CADR machine is not much more difficult than ordinary assembler programming, although there are necessarily many hardware restrictions that have to be obeyed. These include a program counter stack limited to 32 locations, an execution stack frame size limited to 256 slots, and the necessity to check for page faults in connection with memory references.

Some common operations, e.g. *dereferencing*, take more macro-instructions than micro-instructions to express, and so much overhead is saved by having them in microcode. Paradoxically, the space efficiency aspect of the *Construct* algorithm is easier to express in microcode than in Lisp — the microcoded version allocates chunks of memory to make cdr-coded lists, whereas it is unclear how the Lisp version could do that efficiently.

Moreover, micro-programming gives access to useful hardware features of the machine and makes it possible to exploit them directly for supporting logic programming. In particular, the dispatch memory in combination with the byte extractor provide excellent support for the *Unify*$_S$ and *Construct* algorithms.

## 5.5 Storage management

First, we will need a brief outline of the Lisp Machine's storage management scheme. The storage is subdivided into *areas*, each area containing related objects, of any type. Areas are intended to give the user control over the paging behavior of programs. LM-Prolog does so and also uses areas explicitly for allocation of working storage.

Areas are not Lisp nor Prolog objects, but are identified by integers. Areas are passed as arguments to most memory allocation primitives.

The storage of an area consists of one or more *regions*. Each region is a contiguous section of address space with certain homogeneous properties. A *fill-pointer* is associated with each region, and indicates to what extent storage has been allocated in the region. An important property is that a given region cannot contain objects of both of the following categories:

- objects "made out of conses"
  This corresponds to Prolog terms such as variables and conses.
- objects "made out of arrays"
  e.g. arrays, symbols, and instances.

Like areas, regions are not Lisp nor Prolog objects, but are identified by integers. Unlike areas, the user does not have any control over regions and it is not meaningful to pass their identifiers to memory allocation primitives.

The following are the main data areas of the LM-Prolog implementation:

The execution stack.

This is used as the control stack both for Prolog and for Lisp code. It is subdivided into activation frames, each frame holding linkage pointers, arguments, local variables, and working stack, just as for conventional languages. Variables in compiled Prolog code have corresponding slots allocated in the arguments or local variables blocks. The architecture of the CADR machine provides a high-speed memory device for the top 1024 words of the execution stack. The rest of it resides in ordinary virtual memory space.

***structure-area***

This is the name of a global variable whose value identifies the area where the Prolog database is kept.

***prolog-work-area***

This is the name of a special variable which is local to each process running Prolog. It contains a number identifying the area where all dynamic memory allocation takes place, i.e. value cells, object terms, flavor instances, and continuations. Contrary to most Prolog implementations, it is **not** used as a stack that contracts upon backtracking. The reason for this is that due to the complicated memory management scheme of the machine, it is expensive to compute a "stack pointer" $P$ and to reset an area back to $P$. It would entail traversing the area's region list and collecting the fill-pointer of each region.

We have, instead, taken the "brute force" approach of resetting the area only at top level, when a new query is entered. This means that for large problems with a lot of nondeterminism, the machine can run out of storage. We feel, however, that our approach is justified since (*i*) reasonable problems are rather deterministic, (*ii*) the machine has a virtual memory space of at least sixteen million words, (*iii*) one can always resort to using the machine's incremental garbage collector. This last argument only holds for problems that run out of storage because of excessive nondeterminism. For "perpetual" problems with deep recursion, the trail-stack will have to be garbage collected. Algorithms for garbage collecting the trail-stack can be found in [Warren, 1977; Bruynooghe, 1982b].

***trail-array***

This is the name of a special variable which is local to each process running Prolog. It contains the trail-stack. It is arranged so that the value of the special variable *trail* is invisibly linked to the stack-pointer, since compiled code frequently needs access to it. The trail-stack is used by the microcode for trailing variable bindings, and by the run-time system for trailing continuations of things to undo upon backtracking.

# 6. The interpreter

## 6.1 A resolution step

The basic resolution step involving a continuation $C$, a constructed predication $P_C$ and a source clause $P_S \leftarrow Q_S$ is performed as follows:

- The $Unify_S$ algorithm is applied to $P_C$ and $P_S$. This fails or creates a temporary environment $E$.
- In case of success, the *Construct* algorithm is applied to $Q_S$ in $E$ creating $Q_C$, the constructed body of the clause. $E$ is discarded. The resolvent $Q_C$ then has to be proved as a conjunction with continuation $C$.

The backtracking mechanism supporting proofs of conjunctions is discussed in the following section.

## 6.2 Conjunctions

This section contains a natural language paraphrase of the workings of the interpreter. The source listing can be found in Appendix A.

**Prove-conjunction** $Q$ $C$

If the conjunction $Q$ is empty we invoke the continuation $C$. Otherwise we prove the first of $Q$ using its prover with continuation to prove the rest of $Q$ with continuation $C$.

**Try-each-clause** $R$ $C$ $A$ $m$

If the list of alternative clauses $R$ is empty, we fail and return. Otherwise we attempt a resolution step between the continuation $C$, the argument list $A$, and the first of $R$. If it ultimately fails, we unwind the trail-stack back to the stack pointer $m$ and try the rest of $R$. Otherwise we succeed and return.

The implementation technique upon which the interpreter is based was labeled *goal stacking* by Warren in [Warren, 1983] (page 21) and was independently developed by him. Among its advantages, he mentions good paging behavior since there is no reference to a source clause once resolution with it is complete, although structure shared ground subparts could cause some random memory accesses. Another advantage is the fact that all binding environments are local to the resolution steps. A disadvantage though is the unnecessary copying when a clause is entered and fails early in the body.

The interpreter uses a scratch pad vector (maintained in the Lisp variable *vector*) as its binding environment. Since the construction of a body happens immediately after the unification of a head, the vector does not need to be a recursive resource, except in two cases: (*i*) parallel processes may interfere with each other—therefore each process gets its own environment; (*ii*) special objects like constraints and lazy collections may invoke the interpreter recursively during a unification and so must have their own environments too.

## 6.3 Avoiding going through provers

If prove-conjunction encounters a predication whose definition says :if-another-definition :ignore, is interpretive, un-indexed, and does not contain cut, it performs an optimization.

Instead of calling the definition's prover, prove-conjunction immediately proceeds to try the clauses of the definition. A couple of levels of function calling and passing of argument lists is thereby avoided, and prove-conjunction becomes tail-recursive.

## 6.4 Handling of cut

The implementation of interpreted cut uses the non-local exit mechanism *catch and *throw, provided by Lisp machine Lisp: the form

(*catch *tag-form* $form_1$)

is equivalent to just

$form_1$

unless

(*throw *tag-form* $form_2$)

happens inside of the computation of $form_1$, in which case the *catch-form becomes equivalent to $form_2$.

LM-Prolog capitalizes on this mechanism for implementing interpreted cut. Cut's prover is

```
(defun cut-in-system-prover (continuation)
       (*throw *cut-tag* (invoke continuation)))
```

A special variable *cut-tag* is maintained so that a cut can be performed at any time. Namely, an interpretive prover of a predicate $P$ that contains or might contain a cut performs the following extra actions:.

1. It transforms its continuation so that it, when invoked, will bind *cut-tag* to the value it had at entry to $P$;
2. It binds *cut-tag* to something unique;
3. It tries $P$'s clauses inside of a (*catch *cut-tag* ...).

I.e. the prover sets up a "cut point" local to its computation. The following is an example of a prover capable of handling cut:

```
(defun a-in-user-prover (continuation &rest arguments)
  (rest-arg-fixup arguments)
  (let-if (neq (first continuation) 'true)
          ((continuation (continuation (funcall #'invoke-continuation
                                                 continuation
                                                 *cut-tag*))))
          (let ((*cut-tag* continuation))
            (*catch *cut-tag*
              (try-each-interpreted-clause (contents a-in-user-clauses)
                                           continuation
                                           arguments
                                           *trail*)))))
```

Note. The function **invoke-continuation** binds *cut-tag* to its second argument and invokes its first argument. The prover performs a run-time check to see whether it is being called deterministically i.e. whether its continuation is just true, in which case it is not wrapped in an **invoke-continuation** continuation.

# 7. The compiler

## 7.1 Introduction

LM-Prolog's compiler is invoked when the define-predicate option :execution :compiled is used. If the predicate is declared to be :static, which is the default, the Prolog source is translated to a Lisp procedure that becomes the prover of the predicate being defined. The prover constitutes the standardized entrypoint of all predicates. Its first argument is a continuation. The other arguments are the arguments of calling predications.

If the predicate is :dynamic, its prover becomes the same as for interpreted predicates, i.e. a small procedure that tries the clauses one at a time. Here, each clause is compiled to a Lisp procedure that takes two arguments: a continuation and the argument terms passed by the calling predicate. Indexed predicates are treated as dynamic by the compiler.

In the :static case, local tail recursion is removed from the generated code. In both cases, the Lisp procedure is further translated into machine code.

## 7.2 Compiling a predicate

The compiler performs the following optimizations on :static predicates.

### 7.2.1 Arity restriction.

LM-Prolog predicates in general do not have fixed arity. The formal argument list of a predicate will then be a &rest argument, and unification must pull out the individual arguments. Moreover, the predicate-to-predicate calling mechanism causes unbound arguments to be transmitted as dtp-locative pointers, violating the storage conventions mentioned in Section 5.2 when the &rest argument is regarded as a list. A runtime procedure must traverse the &rest argument and change dtp-locative pointers to invisible pointers.

However, if all the clauses of a predicate have heads with the same arity, the predicate can be compiled into a procedure with a fixed number of arguments and no &rest argument. This eliminates the need for "fixing up" the argument list and also reduces the job of unification. Calling the compiled code with the wrong number of arguments will signal an error.

### 7.2.2 Incompatible head detection.

Aided by :argument-list declarations containing '+', the compiler may detect the following situation: ($H_k$ denotes heads of the predicate being compiled.)

> While analyzing $H_i$, no predication $P$ matching $H_i$ matches any $H_j$, $j > i$.

We are describing the situation that the head of clause $i$ is **incompatible** with all heads in subsequent clauses. That is to say, if a predication matches $H_i$, it cannot possibly match any more heads.

When the compiler detects this situation, it "splices in" a (cut) predication right after the '+'-unifications of the head of clause $i$. In particular, it always splices in a (cut) before the body of the last clause. This helps the third :static optimization, discussed in the next subsection.

Section 9.6 discusses a possible generalization of this optimization.

### 7.2.3 Determinacy analysis.

We have mentioned earlier that a predicate may be declared to be deterministic. The knowledge that a predicate is deterministic is used by the interpreter, which always calls deterministic predicates with continuation true, and when compiling predications. The compiler does not always emit calls to deterministic predicates with continuation true, however, since that would interfere with tail recursion optimization.

For a :static predicate, the compiler performs an analysis to see if it is deterministic after the Incompatible Head Detection step. Our analysis is conservative i.e. it deems a predicate deterministic only if it is deterministic, but the converse does not always hold. The code will still run correctly, but its time and space requirements can be suboptimal when the compiler fails to detect determinacy.

Our analysis goes as follows:

1. During the analysis of a predicate $P$, $P$ is assumed to be deterministic.
2. The system uses knowledge about previously compiled predicates and of some built-in predicates, in particular cut.
3. $P$ is deemed non-deterministic if and only if $P$ has some clause that does not have a (cut) in its body or that has a non-deterministic predication after the last (cut) of its body.

For some meta-level predicates that the system knows about, the algorithm is invoked recursively.

Call the outcome of this analysis $C$ and let $D$ denote whether the predicate has been declared to be deterministic. Based on the truth values of $C$ and $D$ we have three cases.

1. $C = D$.
   Compilation proceeds as usual.
2. $C$ but not $D$.
   The compiler automatically emits the determinacy declaration to the benefit of subsequent compilation of predications pertaining to the predicate being compiled.
3. $D$ but not $C$.
   This is the case where the user **forces** a predicate to become deterministic by declaring it so, even though some predications, to the compiler's knowledge, could have more than one solution. In this case, the compiled code must take special action to guarantee determinacy. $P$ compiles to roughly

```
(defun P-in-world-prover (continuation argument_1 ... argument_n)
  (cond ([P compiled as if with continuation true]
         (invoke continuation))))
```

## 7.3 Compiling clauses

Given a predication $P_0$ and the source terms for a set of clauses, the task is to compile the clauses with respect to the predication. This task generalizes to a mechanism which is applied for two different situations in the compiler.

1. At top level of a :static predicate $P$. $P$'s clauses are compiled with respect to what is known about the argument list. The arity range is apparent from the heads of $P$. The argument list declaration may give additional information about whether some arguments can be expected to be bound.
2. When a predication pertaining to an :open predicate is compiled.[1] As much as possible of the unification of the predication and the clause heads is compiled away.

The output of the compilation of a set of clauses is as follows. We have three cases depending on the number of clauses.

- Zero clauses compile to nil.
- One clause compiles to what the resolvent of it and $P_0$ compiles to.
- Two or more clauses compile to

```
(let ((mark *trail*))
 (cond [Compiled resolvent of P0 and first clause.]
       ((progn (untrail mark) nil))
       [Compiled resolvent of P0 and second clause.]
       ((progn (untrail mark) nil))
       ...
       [Compiled resolvent of P0 and last clause.]
       ))
```

The form (untrail x) unwinds the trail-stack back to x. This translation scheme is optimized when the compiler detects that "untrailing" is not going to be needed if a clause has failed, in which case ((progn (untrail mark) nil)) is omitted. This is the case if the clause has a (cut) and the only predications before the first (cut) are predications corresponding to the unification of a '+'-declared argument with a term that is either atomic or a cons of two first-occurrence variables, and predications for initializing local Lisp variables as discussed in the following section. If all ((progn (untrail mark) nil)) are omitted, the binding of mark is also omitted.

## 7.4 Compiling a clause

With respect to a predication $P_0$, a clause is compiled in two steps. First, $P_0$ is *resolved* with it. The resulting *resolvent* is a conjunction of (*i*) goals corresponding to unifying $P_0$ with the head of the clause, and (*ii*) an instance of the body of the clause.

This section describes the "resolution" step. Its purpose is to compute which unification steps, and what variable initializations, have to be performed at runtime.

[1]The normal universe mechanism is disabled here, as the predicate found at compile time expands to in-line code

Given an arbitrary predication and a clause, this step singles out a set of pairs ⟨*variable, term*⟩ that have to be unified at runtime. Those "small" unifications are further compiled with respect to *term* in subsequent steps.

To initialize a variable means to set a local Lisp variable to a newly allocated value cell. Prolog variables compile to local Lisp variables, except if they only occur once or if they can be equated to the car or cdr of a '+'-declared argument. Variable names are allocated in a stack discipline so that the variable names used by one clause are re-used by subsequent clauses. This discipline is recursively applied when open-compiling predications.

Variable initializations may be needed both before and after the unification steps.

"Before" initializations:

> Single-occurrence variables are specially optimized in the implementation so that they do not occupy local Lisp variables. If they occur as top-level arguments in the head, they disappear. Otherwise, they become newly allocated value cells which are passed as arguments or stored inside structures. Now, if a variable only occurs once, and that occurrence is in a predication that is compiled open, the variable may turn out to occur more than once after the opening. The compiler detects this when it performs the resolution step and emits code to initialize such variables prior to the subsequent unification steps.

"After" initializations:

> When a predication is matched against the head of a clause, variables that only occur in the body of the clause are not initialized. The resolution step has to emit code for initializing such variables. A subsequent step moves the code into the compiled code of the body so that the initializations happen as late as possible since (*i*) some initializations will be unnecessary if a predication fails early in the body, (*ii*) the size of continuations depends on how many variables have to be passed to them.

The resolution step is performed as follows:
The compiler unifies the predication with the head of the clause. If the compile-time unification fails, the run-time unification is bound to fail also. If it succeeds, the ⟨*variable, term*⟩ pairs are collected from the trail-array and the body of the clause is constructed in the environment of the unification.

Let $\xi^*$ denote $\xi$ repeated zero or more times. The output from the resolution step, the ***resolvent***, is the symbol impossible, if the compile-time unification failed, or a conjunction

(and (initvar $u$)$^*$ (unify $v$ $t$)$^*$ (initvar $w$)$^*$ . *body*)

where
*body* is the constructed body,
$u \in$ the previously single-occurrence variables that now occur more than once,
$\langle v, t\rangle \in$ the trail-array,
$w \in$ the variables that were allocated by the construction of the body.

Thus it consists of an instance of the body of the resolved clause preceded by *unify* predications, each corresponding to the runtime unification of a variable with a term, and *initvar* predications, each corresponding to the runtime initialization of a variable.

Let us consider a few examples of the described process. Our first example will treat the first clause of the naive-reverse predicate discussed in Appendix C:

```
((naive-reverse (?first . ?rest) ?reverse)
 (naive-reverse ?rest ?rest-reversed)
 (concatenate ?rest-reversed (?first) ?reverse))
```

compiled with respect to the argument list at top level of the predicate:

```
(naive-reverse ?v1 ?v2)
```

yielding the resolvent

```
(and;;no "before" initializations
 (unify ?v1 (?first . ?rest)) ;;a "unify" predication
 (initvar ?rest-reversed) ;;an "after" initialization
 (naive-reverse ?rest ?rest-reversed) ;;part of the body
 (concatenate ?rest-reversed (?first) ?v2) ;;part of the body
 )
```

Note that ?v2 was substituted for ?reverse in the body.

As an example of open-coding, consider the clause

```
((p ?x ?y) (q ?x ?y ?y))
```

compiled with respect to the predication

```
(p ?a ?)
```

This yields the resolvent

```
(and
 (initvar ?g0001);;a "before" initialization, and ?g0001 is marked
 ;;as a multiple-occurrence variable. No "unify" predications,
 ;;no "after" initializations. The body is:
 (q ?a ?g0001 ?g0001))
```

## 7.5 Compiling resolvents

As explained in Section 7.3, resolvents compile to branches of a Lisp cond. Since resolvents are conjunctions of predications, one would expect them to compile to conjunctions of Lisp procedure calls or something similar. In the general case, however, they do not. Resolvents generally compile to constructs that are nested in two ways. The predication (cut) and the necessity to pass continuations attribute to the nesting.

The predication (cut) causes resolvents to compile to nested cond statements, modelling the fact that (cut) commits the execution to a particular path. A resolvent

(and $R_0$ (cut) $R_1$ (cut) ... (cut) $R_n$)

where $R_i$ denotes a sequence of predications not containing (cut), compiles to the cond branch

```
((funcall
  (current-entrypoint $\overline{R_{E0}}$)
  (continuation (true))
  . $\overline{R_{A0}}$)
 (cond ((funcall
         (current-entrypoint $\overline{R_{E1}}$)
         (continuation (true))
         . $\overline{R_{A1}}$)
        (cond (...
               (funcall (current-entrypoint $\overline{R_{En}}$)
                        original-continuation
                        . $\overline{R_{An}}$))))))
```

where $R_{Ei}$ denotes the predicator of $R_i$, $R_{Ai}$ denotes the argument list of $R_i$, current-entrypoint computes what the prover of its argument is. Only the first member of the relations $R_{E0} \ldots R_{En-1}$ is needed, this is enforced by passing (continuation (true)). original-continuation is the name of the continuation argument of the procedure being emitted.

Non-deterministic predications cause resolvents to compile to nested continuations. A resolvent

(and $P_0$ $P_1$ ... $P_n$)

where $P_i$ denotes a non-deterministic predication, compiles to the cond branch

```
((funcall
  (current-entrypoint $\overline{P_{E0}}$)
  (continuation
   (funcall
    (current-entrypoint $\overline{P_{E1}}$)
    (continuation
     ...
     (continuation
      (funcall
       (current-entrypoint $\overline{P_{En}}$)
       original-continuation
       . $\overline{P_{An}}$))
     ...)
    . $\overline{P_{A1}}$))
  . $\overline{P_{A0}}$))
```

where $P_{Ei}$ denotes the predicator of $P_i$ and $P_{Ai}$ denotes the argument list of $P_i$.

Deterministic predications, however, do compile to conjunctions of predicate-to-predicate calls. A resolvent

(and $Q_0$ $Q_1$ ... $Q_n$)

where $Q_i$ denotes a deterministic predication, compiles to the cond branch

```
((and (funcall (current-entrypoint $\overline{Q_{E0}}$)
               (continuation (true))
               . $\overline{Q_{A0}}$)
      (funcall (current-entrypoint $\overline{Q_{E1}}$)
               (continuation (true))
               . $\overline{Q_{A1}}$)
      ...
      (funcall (current-entrypoint $\overline{Q_{En}}$)
               original-continuation
               . $\overline{Q_{An}}$)))
```

where $Q_{Ei}$ denotes the predicator of $Q_i$ and $Q_{Ai}$ denotes the argument list of $Q_i$.

This intuitive account of the compilation process describes how the control structure of Prolog is translated into Lisp. The translation scheme can be made more precise by writing it in terms of Prolog predicates. We need to introduce three predicates that define the mapping from resolvents to cond branches.

### 7.5.1 CR—Compile Resolvent

used as (cr *Resolvent Ante Conse R-Ante R-Conse*) with declarative reading

> (*R-Ante* . *R-Conse*) is the result of compiling *Resolvent* into the cond branch (*Ante* . *Conse*).

could be defined in LM-Prolog as:

```
(define-predicate cr (:options (:deterministic :always))
  (;;Uninstantiated conjunction—call the interpreter
   (cr ?atom ?ante ?conse
       (prove-conjunction-if-need-be ?atom1 (continuation ?ante))
       ?conse)
   (variable ?atom)
   (ct ?atom ?atom1))
  (;;Impossible resolvent—fail
   (cr impossible ? ? nil ()))
  (;;An empty resolvent—base case
   (cr () ?ante ?conse ?ante ?conse))
  (;;A non-empty resolvent—recurse
   (cr (?p . ?ps) ?ante ?conse ?ante2 ?conse2)
   (cr ?ps ?ante ?conse ?ante1 ?conse1)
   (cp ?p ?ante1 ?conse1 ?ante2 ?conse2)))
```

### 7.5.2 CP—Compile Predication

used as (cp *Predication Ante Conse P-Ante P-Conse*) with an analogous declarative reading could be defined in LM-Prolog as:

```
(define-predicate cp (:options (:deterministic :always))
 (;;A (cut) introduces an implication.
  (cp (cut) ?ante ?conse (true) ((cond (?ante . ?conse)))))
 (;;Closed, non-deterministic compilation.
  (cp (?p . ?args) ?ante ?conse
      (funcall (current-entrypoint ?c-p) (continuation ?ante) . ?c-args)
      ?conse)
  (is-predicate ?p :closed :non-deterministic)
  (ct ?p ?c-p)
  (bag-of ?c-args ?c-arg (member ?arg ?args) (ct ?arg ?c-arg)))
 (;;Closed, deterministic compilation.
  (cp (?p . ?args) ?ante ?conse;;The deterministic case.
      (and (funcall (current-entrypoint ?c-p) (continuation (true)) . ?c-args)
           ?ante)
      ?conse)
  (is-predicate ?p :closed :deterministic)
  (ct ?p ?c-p)
  (bag-of ?c-args ?c-arg (member ?arg ?args) (ct ?arg ?c-arg)))
 )
```

### 7.5.3 CT—Compile Term

used as (ct *Term Code*) with declarative reading

*Code* is a Lisp expression that computes *Term*.

## 7.6 Compiling unification

This section describes how *unify* predications are compiled. *Unify* predications look like (unify *variable term*), where *term* is a subterm of a clause head, and are generated by the compiler in the resolution step, as described in Section 7.4. They specify unifications to be done at run time. Their compilation depends on whether there is microcode support for unification (there normally is). Some special optimizations are shared between the two cases, though, when *term* is a variable.

If *term* does not correspond to a local Lisp variable and this is the first occurrence of *term*, we emit:

t

If *term* does compile to a local Lisp variable and this is the first occurrence of *term*, we emit:

(progn (setq $\overline{\textit{term}}$ $\overline{\textit{variable}}$) t)

If this is a subsequent occurrence of *term*, we emit:

(%unify-term-with-term $\overline{\textit{term}}$ $\overline{\textit{variable}}$)

The following three optimizations are performed if *variable* has been '+'-declared and *term* is a non-variable.

If *term* is a symbol, the scheme is optimized to:

(eq $\overline{\textit{variable}}$ $\overline{\textit{term}}$)

If *term* is an atom, we emit:

(equal $\overline{\textit{variable}}$ $\overline{\textit{term}}$)

If *term* is a cons, we emit:

(and (consp $\overline{\textit{variable}}$) $C_a$ $C_d$)

where $C_a$ and $C_d$ denote the compiled unification of the cars and cdrs.

### 7.6.1 When microcode support is available

With microcode support, the default is to emit

(%unify-term-with-template $\overline{\textit{variable}}$ $\overline{\textit{sourceterm}}$)

where *source term* is the source term standing for *term*.

### 7.6.2 When microcode support is not available

Without microcode support, the unification compiles to a nested construct depending on *term*:

If *term* is a first occurrence of a '-'-declared formal argument, we emit

(bind-cell-check $\overline{\textit{term}}$ $\overline{\textit{variable}}$)

if *term* is atomic, we emit

(unify-atom $\overline{\textit{variable}}$ $\overline{\textit{term}}$)

if *term* is a cons, we emit

```
(cond ((uvar-p $\overline{variable}$)
        (bind-cell-check $\overline{variable}$ $\overline{term}$))
       ((consp $\overline{variable}$)
        (and (let ((temp (car $\overline{variable}$)))
               $C_a$)
             (let ((temp (cdr $\overline{variable}$)))
               $C_d$)))))
```

Here, unify-atom is the special case of unification where the second argument is instantiated to an atom, bind-cell-check is the special case of unification where the first argument is uninstantiated or a flavor instance, and uvar-p tests if bind-cell-check is applicable. Note that $\overline{\textit{term}}$ is an expensive computation if *term* is a cons, since it amounts to the construction of a complex term which we do not want to perform if unnecessary.

$C_a$ denotes the compiled unification of temp and the car of *term*. $C_d$ analogously denotes the compiled unification of temp and the cdr of *term*.

Argument-list declarations are used to cut branches of the nested conditionals. Since the size of the emitted code for the case without microcode support grows at least quadratically with the size of *term*, there is a bound on the recursion depth so that all conses deeper than the bound compile to:

(%unify-term-with-term $\overline{variable}$ $\overline{term}$)

### 7.6.3 An Example

The example shows the compilation of the unifications involved with a predicate that unifies two lists. Without microcode support, the predicate

```
(define-predicate concatenate (:options (:argument-list (+ + -)))
  ((concatenate (?first . ?rest) ?back (?first . ?all-but-first))
   (concatenate ?rest ?back ?all-but-first))
  ((concatenate () ?back ?back)))))
```

compiles to

```
(deffun concatenate-in-user-prover
        (continuation ?variable0 ?variable1 ?variable2 &aux ?variable5)
  (let ((g1074 (%dereference ?variable0)))
    (cond (;;((concatenate
           (and (cond ((consp g1074) (and t t)))) ;;(?first . ?rest)
           (and (cond (t ;;?back
                           (setq ?variable5 (%cell0))
                           (bind-cell-check ;;(?first . ?all-but-first))
                             (%dereference ?variable2)
                             (prolog-cons (car g1074) ?variable5))))
                (funcall ...))) ;;(concatenate ...))
          ( ;;((concatenate
           (and (eq g1074 'nil)) ;;()
           (and (progn (bind-cell-check ;;?back ?back)
                         (%dereference ?variable2)
                         (%dereference ?variable1)))
                (invoke continuation))) ;;)
          )))
```

The example exemplifies some optimizations made feasible by the :argument-list declaration:

- The '+'-declared first argument ?variable0 is dereferenced once as opposed to at every occurrence
- The logical variables ?first and ?rest do not need to occupy local variables since they are the car and cdr of ?variable0
- The first and third arguments of the head of the first clause compiled to unconditional code; the second argument was compiled away
- The Lisp constructs (eq ... nil) and (consp ...) compile to particularly efficient machine code sequences

## 7.7 Compiling built-in primitives

Built-in primitives are sometimes called "evaluable predicates". The predicate cp described in Section 7.5 is actually :dynamic: Primitives that need to be compiled specially are conceptually incorporated by asserting "compile methods" as new clauses of cp. In the present implementation, however, each "compile method" is implemented as a Lisp procedure.

Primitives can be deterministic or non-deterministic. It is completely up to the "compile method" to translate predications of the primitive.

For interpreted code to be able to call built-in primitives, they must also be defined as Prolog predicates possessing an extra declaration (:compile-method :intrinsic *procedure-name*), typically with just one clause containing a recursive call to itself.

Unwind-protect, for instance, is defined as:

```
(define-predicate unwind-protect
  (:options (:compile-method :intrinsic ...))
  ((unwind-protect ?do . ?undo) (unwind-protect ?do (and . ?undo))))
```

and has the following compile-method:

```
(unwind-protect ;;The clause name, see section 2.2
 (cp (unwind-protect ?do . ?undo) ?ante ?conse
     (let ((cleanup (continuation (cond ((?undo-ante . ?undo-conse))))))
       (trail cleanup)
       ?do-ante)
     ?do-conse)
 (cp ?do ?ante ?conse ?do-ante ?do-conse)
 (cr ?undo (false) () ?undo-ante ?undo-conse))
```

**Notes.** The form (trail x) *trails* x i.e. adds x to the trail-stack. The predication (unwind-protect ?do . ?undo) compiles just like ?do except that code for trailing a continuation to do ?undo is inserted before the code for ?do. More precisely, ?undo will be exhaustively executed, since it is compiled with continuation (false), causing the compiled code to immediately backtrack after each solution.

Let us consider an example of the unwind-protect mechanism. The predication (assume ?clause) asserts ?clause and retracts ?clause upon backtracking. The predicate assume is defined as

```
(define-predicate assume
 ((assume ?clause) (unwind-protect (assert ?clause) (retract ?clause))))
```

and compiles to

```
(defun assume-in-system-prover (original-continuation ?variable0)
  (let ((cleanup
         (continuation
          (cond ((funcall (current-entrypoint 'retract)
                          (continuation (false))
                          ?variable0))))))
    (trail cleanup)
    (funcall (current-entrypoint 'assert) original-continuation ?variable0)))
```

# 8. The control language

## 8.1 Introduction

The basic computational model of LM-Prolog is the sequential Prolog model. Using certain control language expressions, the user gets access to alternative models, namely functions and variants of producer-consumer relationships. Functions have already been treated in Section 2.1. Three kinds of producer-consumer relationships can be expressed in our language:

1. A producer agent, producing members of a relation, may be connected by unbounded buffers to any number of consumers. This idea is related to the streams of Kahn and McQueen [1977] and to those of Landin [1965]. Streams in the context of logic programming have also been treated by Clark *et al.* [1982], by Bellia *et al.* [1982], and by Hansson *et al.* [1981]. Our producers can be lazy or eager, and optionally produce only unique elements of a relation. We thus have four possibilities, denoted by lazy-bag-of, lazy-set-of, eager-bag-of, and eager-set-of, where 'set' denotes unique elements. With lazy producers, we arrive at a form of lazy evaluation (see Henderson and Morris [1976].)

2. A goal containing an uninstantiated variable $V$ may be suspended until $V$ is instantiated. This can be viewed as the communication of a single value between a producer and a lazy consumer. It is essentially the data-flow computation rule of Dennis [1971], and is denoted by constrain. This computation rule can also be regarded as a dynamic typing facility, since $V$ can be associated with a predicate which is run when $V$ is instantiated, i.e. the range of $V$ is constrained. Constraints as a computational device have been studied by Stallman and Sussman [1977] and others. The integration of constraints into Logic Programming was conceived by Colmerauer [1980].

3. An alternative way of expressing producer-consumer relationships is by variable annotations, as in IC-Prolog [Clark *et al.*, 1982] where a variable occurrence can be marked as a consumer or a producer by a suffix character. Later, consumer annotations have been used in the context of and-parallelism for data-flow synchronization in Shapiro's [1983] Concurrent Prolog, where they are called read-only annotations. Shapiro's interpreter runs in LM-Prolog with read-only annotations written as suffix '?' and supported at the source term and microcode level.

These computational models all capitalize on LM-Prolog's ability to introduce nonstandard or ***noncanonical terms*** (cf. Martin-Löf [1979]) of new datatypes as subflavors of prolog-flavor, as described in Section 5.3. Since this general facility is confined to unification, all the machinery needed by these computational models resides in the :unify operations of the new datatypes, which has the important benefit that all of these models are fully available from compiled code.

Assume, for example, that a relation has three elements $a$, $b$, and $c$. A request to lazily compute this relation leads to the creation of a noncanonical term $I_0$. The unification of $I_0$ triggers the computation of the first element of the relation, i.e. $I$ "reduces" to $a.I_1$, and so on.

The expressive power of lazy and eager collectors is treated in [Kahn, 1984].

## 8.2 Lazy collections

### 8.2.1 Declarative semantics

(LAZY-BAG-OF *bag term . predications*)
(LAZY-SET-OF *set term . predications*)

*Bag* and *set* are, respectively, the list of instantiations of *term* such that *predications* hold, and the elements of *set* are unique.

### 8.2.2 Operational semantics

(LAZY-BAG-OF *bag term . predications*)
(LAZY-SET-OF *set term . predications*)

unify *bag* or *set* with the list of all instantiations of *term* such that *predications* hold. lazy-set-of suppresses copies.

The list is built up only when needed. The need can arise from a unification of the list with an instantiated term. The need can also arise if the list is passed to Lisp or printed.

### 8.2.3 An example

The predication (prime ?p) as defined below is true if ?p is a prime number. Upon backtracking, ?p is unified with successive primes numbers. The program is a version of the Sieve of Eratosthenes. It represents lists of numbers as lazy collections. The list 2.3.4.... is transduced to 3.5.7.... which is transduced to 5.7.11.... and so on.

```
(define-predicate prime;;the second argument is a list of numbers
  ((prime ?prime);;and defaults to 2.3.4....
   (lazy-bag-of ?ints ?int (integer-from ?int 2))
   (prime ?prime ?ints))
  ((prime ?prime (?prime . ?)))
  ((prime ?prime (?seed . ?rest))
   (lazy-bag-of ?sifted ?not-multiple (not-multiple ?not-multiple ?seed ?rest))
   (prime ?prime ?sifted)))


(define-predicate integer-from
  ((integer-from ?i ?i))
  ((integer-from ?i+ ?i)
   (sum ?i+1 ?i 1)
   (integer-from ?i+ ?i+1)))


(define-predicate not-multiple
  ((not-multiple ?number ?seed (?number . ?))
   (quotient ?q ?number ?seed);;fixnum arithmetics
   (cannot-prove (product ?number ?q ?seed)))
  ((not-multiple ?number ?seed (? . ?numbers))
   (not-multiple ?number ?seed ?numbers)))
```

### 8.2.4 Implementation

There are many opportunities for compile-time optimizations of constructs involving lazy-bag-of or lazy-set-of, which will be covered in the following subsection. Let us cover the general case of lazy-bag-of first. Lazy-set-of is similar except for the suppressing of copies.

We use the flavor lazy-collection which has subflavors lazy-bag and lazy-set with instance variables

| | |
|---|---|
| collection | the lazy list being built up |
| generator | a continuation to invoke to get more solutions. The continuation will resume a *stack group* where the state of the lazy collection is kept. Resuming a stack group is a relatively expensive operations. |
| constructor | a continuation to invoke to instantiate a solution |

and :unify method defined as

```
;To unify x with a lazy collection,
;unify x with the :ordinary-term of the collection.
(defmethod (lazy-collection :unify) (other-term)
  (unify (send self ':ordinary-term) other-term))


(defmethod (lazy-bag :ordinary-term) ()
  (cond ((variable-boundp collection)
         collection);;collection has already been computed
        ((null (setq generator (invoke generator)))
         (setq collection ())) ;;() for no more solutions
        (t (setq collection
                 (prolog-cons ;;a cons of a solution instance
                   (invoke constructor) ;;and a new lazy bag
                   (make-instance-in-area *prolog-work-area* 'lazy-bag
                    ':generator generator ':constructor constructor))))))
```

Instances of lazy-set have one more instance variable containing list of answers computed so far. The flavor's :ordinary-term suppresses solutions already in the list.

### 8.2.5 Optimizations

There are several opportunities to optimize the compilation of a lazy collection. Let $S$ stand for

(LAZY-BAG-OF *bag term* . *predications*)

- If *bag* is ?, $S$ compiles to nothing and a warning is emitted.
- If *bag* is (), $S$ compiles like (cannot-prove . *predications*).
- If *bag* is ($x$ . ?) or if *predications* are known to be deterministic, *predications* are compiled with continuation true since only one solution is ever needed.
- In the general case, $S$ compiles to code using lazy-collection objects as described in the previous subsection.

## 8.3 Eager collections

### 8.3.1 Declarative semantics

(EAGER-BAG-OF *bag term . predications*)
(EAGER-SET-OF *set term . predications*)

*Bag* and *set* are, respectively, the list of instantiations of *term* such that *predications* hold, and the elements of *set* are unique.

### 8.3.2 Operational semantics

(EAGER-BAG-OF *bag term . predications*)
(EAGER-SET-OF *set term . predications*)

unify *bag* or *set* with the list of all instantiations of *term* such that *predications* hold. eager-set-of suppresses copies.

The list is built up in a separate process in parallel with the computation that invoked the eager collection. If the list takes part in unification, the unifier (or rather the proper :unify method) synchronizes the processes by causing the current process to wait until the collection has terminated. Synchronization is also needed if the list is passed to Lisp or printed.

### 8.3.3 An example

The Sieve of Eratosthenes can be run in eager mode, i.e. representing lists of integers as eager collections. This application of eager collections however leads to an explosion of resource requirements, since the collections compete for resources. The eager version is much less efficient than the lazy version.

### 8.3.4 Implementation

We use the flavor eager-collection which has subflavors eager-bag and eager-set with instance variables

| | |
|---|---|
| process | the Lisp Machine process that this bag will run in |
| value | the eager list being built up |
| status | :finished, :running, or :stopped |
| inferiors | eager bags created by this eager bag |
| constructor | a continuation to invoke to instantiate a solution |

and :unify method defined as

```
;To unify x with an eager collection,
;wait until it has finished its computation, and unify x with
;the list of instantiated solutions.
(defmethod (hurried :unify) (other)
  (selectq status
    (:finished (unify value other))
    (:running (process-wait "Eager value" self ':not-hurried-value)
              (send self ':unify other))))


(defmethod (hurried :not-hurried-value) ()
  (neq status ':running))
```

Instances of eager-set have one more instance variable containing list of answers computed so far. The flavor's :ordinary-term suppresses solutions already in the list.

#### 8.3.5 Optimizations

The same optimizations as for lazy collections apply.

## 8.4 Constraints

### 8.4.1 Declarative semantics

The predication

(CONSTRAIN *variable predication*)

is equivalent to just *predication.*

### 8.4.2 Operational semantics

If variable is bound when the predication (CONSTRAIN *variable predication*) is encountered, *predication* is invoked. Otherwise, the execution of *predication* is postponed until *variable* gets bound. This is realized by binding *variable* to a constraint object which invokes *predication* if subsequently unified with a non-variable.

### 8.4.3 An example

The predicate send-more-money defined below solves the cryptarithmetics equation "SEND + MORE = MONEY", where each letter stands for a unique digit. The algorithm amounts to setting up six constraints on the unknowns (one for each of the 1-, 10-, 100-, and 1000-columns of the additions, and the two constraints $S > 0$ and $M > 0$). Finally the eight unknowns must be a subset of a permutation of the numbers $0 \ldots 9$.

The control structure of this example can be described as a coroutining relationship between a generator (a producer) (the permutation) and a test (a consumer) (the equation). The example illustrates a general technique for writing clear programs that exhibit coroutining behavior between a generator and a test. We have written a program solving the 8 Queens problem in the same style.

In terms of slogans, we have turned "generate and test" into "constrain and generate".

```
(define-predicate send-more-money
  ((send-more-money (?s ?e ?n ?d) (?m ?o ?r ?e) (?m ?o ?n ?e ?y))
   (prolog:constrain ?s (> ?s 0))
   (prolog:constrain ?m (> ?m 0))
   (prolog:constrain ?e (add-two-with-carry ?c1 ?y ?d ?e 0))
   (prolog:constrain ?r (add-two-with-carry ?c2 ?e ?n ?r ?c1))
   (prolog:constrain ?o (add-two-with-carry ?c3 ?n ?e ?o ?c2))
   (prolog:constrain ?m (add-two-with-carry ?m ?o ?s ?m ?c3))
   (sub-permutation (0 1 2 3 4 5 6 7 8 9) (?y ?d ?e ?n ?r ?o ?s ?m))))

(define-predicate add-two-with-carry
  ((add-two-with-carry 0 ?sum ?x ?y ?c)
   (sum ?sum ?x ?y ?c))
  ((add-two-with-carry 1 ?sum ?x ?y ?c)
   (sum ?sum ?x ?y ?c -10)))

(define-predicate sub-permutation
  ((sub-permutation ? ()))
  ((sub-permutation ?all (?x . ?but-one-perm))
   (extraction ?all ?x ?but-one)
   (sub-permutation ?but-one ?but-one-perm)))

(define-predicate extraction
  ((extraction (?x . ?l) ?x ?l))
  ((extraction (?x . ?all) ?y (?x . ?some))
   (extraction ?all ?y ?some)))
```

### 8.4.4 Implementation

Constraint objects are implemented as instances of the flavor constraint with instance variables

cell The value cell of the constrained variable.

constraints
A list of predications constituting the conjunction of the constraints that have been put on a variable.

The :unify method splits into three cases.

1. Both the current object and the argument of the method are constraint objects with unbound cells.
   This means that the new constraint must be combined with the current one. Logically it is the same as adding it to the conjunction of constraints. Constraints can be combined more intelligently, however, by the user providing assertions. The constraint combination mechanism invokes the predication (COMBINE-CONSTRAINTS *combined current new*). If it succeeds and instantiates *combined* to ((false)), the unification method fails. If it succeeds otherwise, *combined* is used. If it fails, (*new* . *current*) is used.

2. If the current object is unbound and the argument is a non-variable, its cell is trailed and is set to the argument and the constraints are run and must succeed for the unification to succeed. Upon backtracking, the cell of the current object is made unbound.

3. If the current object is bound, the bound value is unified with the argument.

## 8.5 Concurrent Prolog

### 8.5.1 Introduction

Concurrent Prolog [Shapiro, 1983] is a concurrent logic programming language with *guarded* clauses to control search, and *read-only annotations* for data-flow synchronization of concurrent subgoals. Guards originate from Dijkstra [1976] and Hoare's CSP [1978] as a notation for committed-choice non-determinism. They were subsequently adapted for logic programming in the relational language of Clark and Gregory [1981] and its descendant PARLOG [Clark and Gregory, 1983].

Shapiro's interpreter was ported to LM-Prolog by Kenneth Kahn. Our codes for source terms were extended to support read-only annotations, involving changes to the $Unify_S$ and *Construct* algorithms.

### 8.5.2 Implementation of read-only objects

A read-only object is an instance of the flavor read-only-variable which has the instance variable cell. A read-only object unifies with another object $Y$ only if cell is bound to a non-variable that unifies with $Y$ or if $Y$ is a variable. Read-only objects are used as a synchronization primitive in the language. The language admits concurrent execution of conjuncts, and the primitive is used for letting a conjunct wait for another to bind a variable. The implementation re-tries any unification that failed due to read-only variables, and so simulates concurrency in a busy-wait fashion.

### 8.5.3 Read-only annotations

Variables may be given read-only annotations written as a postfix '?'. A read-only annotation $u?$ may be considered as syntactic sugar for a newly constructed read-only object with $u$ being the value of cell. In the implementation, however, the various uses of read-only annotations are optimized so that read-only objects are allocated only when needed.

### 8.5.4 Unification

Shapiro wrote a meta-level unifier in Prolog that recognized read-only annotations as a special functor. We have included read-only annotations in the basic unification algorithms at the firmware level. We use the general flavor mechanism for unifying read-only objects.

# 9. Conclusion and Open Questions

## 9.1 Introduction

We have implemented the world's second complete Prolog compiler, comparable in performance to the first one, the one for DEC-10 Prolog [Warren, 1977, 1980]. Whereas the programming environment of DEC-10 Prolog consists of a debugging package only, our implementation combines the power of the logic programming formalism with the extremely rich and powerful programming environment of personal Lisp machines. We have thus achieved a unique tool with which, for the first time, serious problems can be attacked.

The viability of continuation techniques for implementing logic programming in a functional setting has been demonstrated. A new technique based on a 'genetic code' for terms has been used for constructing complex terms in a demand-driven fashion during unification. A microcoded unifier using dispatch hardware makes the genetic code technique feasible. The technique has been generalized to handle variable annotations.

Our implementation is open-ended since (*i*) LM-Prolog can easily call Lisp functions and (*ii*) there is a standard protocol for extending unification for user-defined datatypes. This second facility has enabled us to incorporate alternative computational models such as demand-driven and parallel computation and constraints without restricting their use to the interpreter.

## 9.2 Comparison with other work

DEC-10 Prolog [Warren, 1977, 1980] is the most generally known implementation of Prolog. Warren defines a virtual Prolog machine, "PLM", based on structure sharing and especially well suited for the DEC-10 architecture. PLM is a "register" machine incorporating instructions for unification, stack space allocation, indexing, procedure calls, cut etc. Our compiler translates Prolog predicates to Lisp procedures. We handle unification and term construction in microcode. However, the abstract Lisp machine emulates Prolog's control structure.

In a recent paper [Warren, 1983], a new abstract machine for Prolog called the "New Engine" is defined. It is a major revision of the PLM and is based on structure copying. Not surprisingly, it has important similarities with LM-Prolog, but also significant differences. The following (incomplete) table summarizes a comparison between PLM, New Engine, and LM-Prolog.

| PLM | New Engine | LM-Prolog |
|---|---|---|
| "Register" machine based on local and global stacks, and trail. | "Register" machine based on stack, heap, trail, and PDL for unification. | "Stack" machine based on stack, heap, trail, and PDL. |
| Variables classified as void, temporary, local, and global. | Variables classified as void, temporary, safe permanent, and unsafe permanent. | Variables classified as void, temporary, and permanent. |
| Local and global environments. | Local environment (not necessarily on top of stack), local and global value cells. | Local environment (always on top of stack, copied if need be), global value cells. |
| Unify instructions can trigger backtracking. | Unify instructions can trigger backtracking. | Unify instructions return truth value which is tested by subsequent code. |
| "Code pointer" into the middle of compiled code. | "Code pointer" into the middle of compiled code. | "Code pointer" corresponds to a compiled Lisp procedure (sometimes embedded). |
| Head arguments compile both to skeleton literals and to instructions, compiled code possibly of exponential size. Mode declarations reduce the size. | Head arguments compile to instructions. Compiled code of linear size. No mode declarations. | Head arguments compile to encoded source terms that are traversed by firmware. Compiled code of linear size. Mode declarations may unfold one level of the encoded source terms. |
| Conditional trailing of bound variables. | Conditional trailing of bound variables. | Unconditional trailing of bound variables (except of first occurrences). |
| Body arguments compile to skeleton literals. | Body arguments compile to construct instructions. | Body arguments compile to construct instructions (cons, list etc.). |
| Runtime pruning of deterministic computations. | Runtime pruning of deterministic computations. | Compile-time detection of determinacy. No direct access to latest backtrack point. |
| No unfolding of predicates and arguments of meta-level predicates are not compiled. | ? | Optional unfolding of predicates. Arguments of meta-level predicates are compiled. |

Since Prolog's control structure is emulated by the abstract Lisp machine, both the PLM and the New Engine are more optimal for executing standard Prolog programs. It is not so clear, however, how some of the important language extensions of LM-Prolog could be imported into the PLM or the New Engine.

POPLOG [Mellish and Hardy, 1984] is a trilingual programming environment supporting Prolog, Lisp, and POP-2. The Prolog part is compiler-based using a variant of the continuation passing technique. In particular, their unifier takes a continuation and they do not use determinism for compile-time optimization of the control structure.

The Lisp-based Prolog implementations are too numerous to be treated in detail. We will restrict our survey to those having some outstanding feature:

- QLOG [Komorowski, 1980] was perhaps the first Lisp-based interpreter. It was based on structure sharing and defined Prolog predicates as Lisp *fexprs*. Komorowski was the first to point out the potential of inheriting a programming environment by embedding Prolog in Lisp.
- LogLisp [Robinson and Sibert, 1980] is an interpreter incorporating breadth-first and heuristic search mode and a notion of reducibility: Every term or predication that can be interpreted as a Lisp form is evaluated as such before resolution happens.
- Prolog/KR [Nakashima, 1982] has a number of features in common with LM-Prolog e.g. arbitrary arity, multiple worlds, and coroutines. Nakashima has argued for a list-based syntax since it provides equivalence between program and data and thus ease of using program manipulating programs such as structure oriented editors.
- Foolog [Nilsson, 1983b] is a beautiful example of a very concise system consisting of an interpreter written in MacLisp together with a compiler written in Foolog.
- Metalog [Dincbas, 1983] uses a meta-program for controlling the execution, rather than the default control of Prolog.
- eu-Prolog [Kohlbecker, 1984] is an interpreter written in Scheme based on *failure continuations*: A successful result of a procedure call is a pair of an environment and a failure continuation. When invoked, the continuation returns alternatives. The technique of failure continuations is discussed in [Kahn and Carlsson, 1984].

Bruynooghe [Bruynooghe, 1982a] and Mellish [Mellish, 1982] have made the structure copying technology generally known, and have made important contributions to it. The typing scheme of the present work enabling (*i*) optimization of first and single variable occurrences and (*ii*) the use of dispatch hardware has however to our knowledge not been published before.

J.F. Nilsson [Nilsson, 1983a] hints at continuation-based compilation of Prolog into Pascal including determinacy optimizations and data-flow analysis but does not give any algorithms. His work concentrates on a polymorphic type system for Prolog.

## 9.3 Experiences, trials, and errors

An early version of the compiler was based on structure sharing. It used stack groups (coroutines) to implement backtracking rather than continuations. Although stack groups could be optimized away in deterministic cases, compiled code generally suffered from expensive stack group switches.

An early version of the interpreter was also based on structure sharing, It had a typing scheme, different from the present source terms, with separate type codes for special objects rather than falling back on message passing to flavor instances. It used failure continuations.

These early attempts are described in more detail in [Kahn and Carlsson, 1984].

## 9.4 Performance

Since the implementations mentioned in the previous section, performance of compiled code has improved by an order of magnitude and of interpreted by a factor of five.

The notorious Prolog benchmark is the "naïve reversal" of a list 30 long, as defined by the program

```
(define-predicate concatenate (:options (:argument-list (+ + -)))
  ((concatenate (?first . ?rest) ?back (?first . ?all-but-first))
   (concatenate ?rest ?back ?all-but-first))
  ((concatenate () ?back ?back)))
(define-predicate naïve-reverse (:options (:argument-list (+ -)))
  ((naïve-reverse (?first . ?rest) ?reverse)
   (naïve-reverse ?rest ?rest-reversed)
   (concatenate ?rest-reversed (?first) ?reverse))
  ((naïve-reverse () ())))
```

This runs compiled in 48 msec and interpreted in 275 msec on a CADR. It takes 498 inferences to solve the problem suggesting a LIPS (Logical Inferences Per Second) rate of just over 10,000 for compiled code and 1800 for interpreted. A more comprehensive set of Prolog benchmarks is yet to be devised.

## 9.5 Future Enhancements

There is still room for enhancements to the compilation process, for example

- improved clause selection
- detection of common subterms
- functional nesting of computations
- open compilation of recursive predicates
- micro-compilation

### 9.5.1 Improved clause selection

The clause indexing mechanism invoked by the :indexing-patterns option is very accurate but incurs too much overhead to be used for predicates that have less than several dozens of clauses. For unindexed predicates, one can get some selection by the Incompatible Head Detection optimization of the compiler. One problem with the current version of the compiler is that all information from the compilation of one clause is forgotten when the next clause is compiled. The compiler could probably do better e.g. by factoring out tests or subterms that are common to more than one clause head. For predicates with variable arity, it cound emit code to index on the number of arguments. Or it could use '+' declarations to compile into a sort of discrimination set for selecting clauses.

In contrast, Dec-10 Prolog has a very efficient mechanism using special indexing instructions compiled in-line. Those techniques could also be used, but we have the problem that most "constants" in the Lisp machine are susceptible to garbage collection, and so there is a problem of keeping hash tables and the like up to date. This problem accounts for much of the overhead of the present implementation of indexed predicates.

### 9.5.2 Detection of common subterms

This corresponds to the "detection of common subexpressions" problem in compiler design theory. See e.g. [Aho and Ullman 1977].

There is an efficiency problem in the adopted structure copying solution: If a non-ground cons subterm occurs more than once in a clause, then each source code occurrence of it will correspond to a making at run time of an object term, instead of using the term constructed at the first occurrence. One would like to construct just one copy and reuse it for each occurrence.

In a structure sharing implementation this is much less of a problem. Not detecting common subterms leads only to an increased size of compiled predicates. It does not affect the size of run time structures.

#### 9.5.3 Functional nesting of computations

Computations like

(and (sum ?sum ?i ?j) (product ?product ?sum ?k))

do not compile very cleverly, even though sum and product compile open. The intermediate variable, ?sum, is allocated as a value cell. It is first unified with the sum of ?i and ?j. Then its value is multiplied with ?k. Clearly, one would like the compiler to detect this situation and compile the construct to something like

(unify $\overline{?product}$ (times (plus $\overline{?i}$ $\overline{?j}$) $\overline{?k}$))

#### 9.5.4 Open compilation of recursive procedures

Currently, only non-recursive predicates can be compiled open. Tail-recursive predicates could readily be compiled open as well, if knowledge about tail recursion were introduced into the compiler. Tail recursion optimization is currently performed in a post-compilation step.

#### 9.5.5 Micro-compilation

The Lisp Machine has a micro-compiler that transforms machine code to microcode, although with several restrictions and limitations. It was applied to the output of the Prolog compiler for the Naïve-Reverse benchmark, resulting in a factor of 2 speed increase. The figure could be improved somewhat since memory allocation does not micro-compile very cleverly.

A much more drastic approach would be to not use Lisp as the intermediate language of the translation process, but rather use one of the abstract instruction sets that have been proposed in the literature. This instruction set could then be compiled to microcode, or emulated, or both. The problem remains as to how the resulting code would be able to coexist with Lisp-based software. Clearly there would have to be interfaces between the two languages.

## 9.6 Future Research

The Horn clause subset of predicate logic and its realization as a programming language — Prolog — is an extremely powerful computational formalism and language. Indeed, Prolog is one of most powerful programming languages ever conceived and is spreading rapidly as we are now experiencing a virtual boom of activity in the logic programming field.

Even though Prolog has been such a success and has so many nice properties, the language has its drawbacks, some of which are serious. Fears have even been expressed that Prolog might become the Fortran of logic programming exactly because it is spreading so rapidly and has drawbacks such as:

Poor control language.

> The more widely spread dialects of Prolog have but one control directive — the cut. Several authors have compared the cut with the goto statements of imperative languages. Clearly, the cut is very unintuitive and hard to learn. Some uses of it can destroy the declarative reading of programs.
>
> Our extensions to Prolog's control language have rendered cut virtually obsolete.

Restricted logic.

> Since Prolog is based on the Horn clause formalism, the language cannot handle explicitly quantified or negated statements nor implications. "Forall" computations can awkwardly be rewritten in terms of set-of, and for negations one has to resort to the more limited negation as failure. Specifications and definitions that are "naturally" formulated in predicate logic have to be transformed to Horn clauses. The task of transformation is by no means trivial although there exist semi-automatic tools.

Whether a richer subset of predicate logic is feasible as a programming language is a most interesting question for future research. Promising work in that direction has already been made. In [Hansson *et al.*, 1981], the viability of natural deduction as opposed to resolution is demonstrated as a basis for logic programming. The proposed language is a superclass of Horn clauses such that the *definiens* can contain arbitrary logic formulas. In [Haridi, 1981], proof strategies for the logic programming system are presented.

Several questions remain to be solved before the system can be used as a practical language:

Control strategies.

> Haridi's proof strategies essentially omit the choice of control rules. In order for the system to be used as a programming language, most of them must be fixed, whereas some should be programmable in a control language.

Abstract machine.

> Recently, an abstract machine for the system was proposed [Haridi and Sahlin, 1983]. The machine resembles the SECD machine for functional programming [Landin, 1963] and describes permissible state transitions. The machine is rather high-level, treating e.g. unification (or its equivalent in a natural deduction setting) as an atomic operation, and needs to be much elaborated if the goal is an efficient implementation.

Compiler.

> Related to the previous item is the construction of a compiler for the language.

## 10. Acknowledgments

The author would like to thank his colleagues at UPMAIL for many useful discussions. Special thanks to Ken Kahn, who was the co-designer and co-implementor and a rich source of ideas and suggestions; to Martin Nilsson, for discussions about many important implementation issues; and to Åke Hansson and my advisor Sten-Åke Tärnlund, who spent much time reading drafts and suggesting improvements.

# 11. References

[Aho and Ullman, 1977] Aho A.V., Ullman J.D., "Principles of Compiler Design", Addison-Wesley, Amsterdam.

[Battani and Meloni, 1973] Battani G., Meloni H., "Interpreteur du Langage de Programmation PROLOG", Groupe d'Intelligence Artificielle, Université Aix-Marseille II.

[Bellia *et al.*, 1982] Bellia M., Degano P., Levi G., "The Call by Name Semantics of a Clause Language with Functions", in *Logic Programming*, (eds. Clark K., Tärnlund S.-Å.), Academic Press, London.

[Boyer and Moore, 1972] Boyer R.S., Moore J.S., "The sharing of structure in theorem proving programs", Machine Intelligence 7, (eds. Meltzer, Mitchie), Edinburgh UP.

[Bruynooghe, 1982a] Bruynooghe M., "The Memory Management of Prolog Implementations", in *Logic Programming*, (eds. Clark K., Tärnlund S.-Å.), Academic Press, London.

[Bruynooghe, 1982b] Bruynooghe M., "A note on garbage collection in Prolog interpreters", *First International Logic Programming Conference*, Marseille.

[Carlsson and Kahn, 1983] Carlsson M., Kahn K.M., "LM-Prolog User Manual", UPMAIL, Uppsala University, Technical Report No. 24.

[Carlsson, 1984] Carlsson M., "On Implementing Prolog in Functional Programming", UPMAIL Technical Report No. 5B, Uppsala University, and in *Proc. 1984 International Symposium on Logic Programming*, Atlantic City NJ, and to appear in *New Generation Computing* Vol. 2 No. 4, Ohmsha, Ltd. and Springer-Verlag, Tokyo.

[Clark, 1978] Clark K.L., "Negation as failure", in *Logic and Data Bases*, (eds. Gallaire H., Minker J.), Plenum Press, New York NY, 1978, pp. 293-322.

[Clark and Gregory, 1981] Clark K.L., Gregory S., "A relational language for parallel programming", in *Proc. Conf. on Functional Languages and Computer Architectures*, ACM, New York, pp. 171–178.

[Clark *et al.*, 1982] Clark K.L., McCabe F.G., Gregory S., "IC-Prolog Language Features", in *Logic Programming*, (eds. Clark K., Tärnlund S.-Å.), Academic Press, London.

[Clark and Gregory, 1983] Clark K.L., Gregory S., "PARLOG: A Parallel Logic Programming Language", Research Report DOC 83/5, Dept. of Computing, Imperial College, London.

[Clark and McCabe, 1984] Clark K.L., McCabe F., "micro-Prolog", Prentice-Hall, London.

[Colmerauer, 1980] Colmerauer A., "PROLOG and the Infinite Trees", Groupe d'Intelligence Artificielle, Université d'Aix-Marseille II, and in *Logic Programming*, (eds. Clark K., Tärnlund S.-Å.), Academic Press, London.

[Dennis, 1971] Dennis J., "On the Design and Specification of a Common Base Language", in *Computers and Automata*, Brooklyn Polytechnic Institute.

[Dijkstra, 1976] Dijkstra E.W., "A Discipline of Programming", Prentice-Hall, Englewood Cliffs NJ.

[Dincbas, 1983] Dincbas M., Le Pape J.-P., "Nouvelle implementation de Metalog", in *Programmation en logique*, (ed. Dincbas M.), Actes du Séminaire 1983, Perros-Guirec, 22-23 Mars 1983.

[Emden and Lloyd, 1984] van Emden M.H., Lloyd J.W., "A Logical Reconstruction of Prolog II", *Proc. Second International Logic Programming Conference*, Uppsala.

[Fages, 1983] Fages F., "Note sur l'unification des termes de premier ordre finis et infinis", in *Programmation en logique*, (ed. Dincbas M.), Actes du Séminaire 1983, Perros-Guirec, 22-23 Mars 1983.

[Galler and Fisher, 1964] Galler B.A., Fisher M.J., "An Improved Equivalence Algorithm", *Communications of the ACM*, Vol. 7 No. 5, pp. 301-303.

[Greenblatt, 1974] Greenblatt R., "The Lisp Machine", MIT AI Lab Working Paper 79, Cambridge MA.

[Hansson *et al.*, 1981] Hansson Å., Haridi A.S., Tärnlund, S-Å., "Properties of a Logic Programming Language", Technical Report No. 8, UPMAIL, Uppsala University, and in *Logic Programming*, (eds. Clark K., Tärnlund S.-Å.), Academic Press, London.

[Haridi, 1981] Haridi A.S., "Logic Programming based on a Natural Deduction System", Tecn.D. thesis, TTDS, Royal Institute of Technology, Stockholm.

[Haridi and Sahlin, 1983] Haridi A.S., Sahlin D., "Evaluation of Logic Programs based on Natural Deduction", TRITA-CS-8305, TTDS, Royal Institute of Technology, Stockholm.

[Henderson and Morris, 1976] Henderson P., Morris J.H., "A lazy evaluator", *Proc. 3rd ACM Symposium on POPL*, pp. 95–103.

[Henderson, 1980] Henderson P., *Functional Programming, Application and Implementation*, Prentice-Hall, London.

[Hoare, 1978] Hoare C.A.R., "Communicating sequential processes", *Communications of the ACM*, Vol. 21 No. 8, pp. 666–677.

[Huet, 1976] Huet G., "Résolution d'équations dans des langages d'ordre $1,2,\ldots\omega$" Thèse d'Etat, Université de Paris VII.

[Kahn and McQueen, 1977] Kahn G., McQueen D.B., "Coroutines and networks of parallel processes", in *Proc. IFIP 1974*, Elsevier North Holland, Amsterdam.

[Kahn, 1984] Kahn K.M., "A Primitive for the Control of Logic Programs", in *Proc. 1984 International Symposium on Logic Programming*, Atlantic City NJ.

[Kahn and Carlsson, 1984] Kahn K.M., Carlsson M., "How to Implement Prolog on a Lisp Machine", in *Implementations of Prolog*, (ed. Campbell J.A.), Ellis Horwood Ltd., Chichester.

[Knight *et al.*, 1981] Knight T.F.Jr., Moon D.A., Holloway J., Steele G.L.Jr., "CADR", MIT AI Laboratory Memo 528, Cambridge MA.

[Kohlbecker, 1984] Kohlbecker E., "eu-Prolog", Indiana University Computer Science Department, Technical Report No. 155.

[Komorowski, 1980] Komorowski H.J., "QLOG — The software for Prolog and Logic Programming", in *Logic Programming*, (eds. Clark K., Tärnlund S.-Å.), Academic Press, London.

[Kowalski, 1974] Kowalski R., "Predicate Logic as Programming Language", *Proc. IFIP 1974*, Elsevier North Holland, Amsterdam, pp. 569-574.

[Landin, 1965] Landin P., "The Correspondence between ALGOL 60 and Church's lambda notation. Part I", *Communications of the ACM* Vol. 8, No. 2.

[Martelli and Montanari, 1976] Martelli A., Montanari U., "Unification in linear time and space", University of Pisa, Internal Report B76-16.

[Martin-Löf, 1979], Martin-Löf P., "Constructive Mathematics and Computer Programming, Dept. of Mathematics, University of Stockholm.

[Mellish, 1982] Mellish C.S., "An alternative to structure sharing in the implementation of a PROLOG-interpreter", in *Logic Programming*, (eds. Clark K., Tärnlund S.-Å.), Academic Press, London.

[Mellish and Hardy, 1984] Mellish C.S., Hardy S., "Integrating Prolog in the POPLOG environment", in *Implementations of Prolog*, (ed. Campbell J.A.), Ellis Horwood Ltd., Chichester.

[Moon *et al.*, 1983] Moon D., Stallman R.M., Weinreb D., "Lisp Machine Manual", MIT AI Laboratory, Cambridge MA.

[Morris, 1980] Morris F.L., "On List Structures and Their Use in the Programming of Unification", School of Computer and Information Science, Syracuse University.

[Nakashima, 1982] Nakashima H., "Prolog/KR Language Features", *Proc. First International Logic Programming Conference*, Marseille.

[Nilsson, 1983a] Nilsson J.F., "On the Compilation of a Domain-based PROLOG", *Proc. IFIP 1983*, Elsevier North Holland, Amsterdam, pp. 569-574.

[Nilsson, 1983b] Nilsson M., "Foolog — A Small and Efficient Prolog Interpreter", Technical Report No. 20, UPMAIL, Uppsala University.

[Nilsson, 1984] Nilsson M., "The Importance of Determinacy Declarations in Prolog implementations", Technical Report No. 26, UPMAIL, Uppsala University.

[Paterson and Wegman, 1976] Paterson M.S., Wegman M.N., "Linear unification", *Proc. 8th ACM Symposium on Theory of Computation*, pp. 181–186.

[Risch, 1973] Risch T., "REMREC - a program for automatic recursion removal in LISP", report no. DLU 73/24, Datalogilaboratoriet, Uppsala University.

[Robinson, 1965] Robinson J.A., "A Machine-oriented Logic Based on the Resolution Principle", *Journal of the ACM*, Vol. 12, no. 1.

[Robinson, 1976] Robinson J.A., "Fast unification", *Proc. Conf. Mechanical Theorem Proving*, Mathematisches Forschungsinstitut Oberwolfach.

[Robinson and Sibert, 1980] Robinson J.A., Sibert E.E., "Logic Programming in LISP", School of Computer and Information Science, Syracuse University.

[Shapiro, 1983] Shapiro E., "A Subset of Concurrent Prolog and Its Interpreter", ICOT Technical Report, TR-003, ICOT, Tokyo, 1983.

[Stallman and Sussman, 1977] Stallman R.M., Sussman G.J., "Forward reasoning and dependency-directed backtracking in a system for computer-aided circuit analysis, in *Artificial Intelligence* Vol. 9, pp. 135–196.

[Strachey and Wadsworth, 1974] Strachey C., Wadsworth C.P., "Continuations — A Mathematical Semantics For Handling Full Jumps", Programming Research Group, Oxford University.

[Tarjan, 1975] Tarjan R.E., "Efficiency of a Good but Not Linear Set Union Algorithm", *Journal of the ACM* Vol. 22 No. 2, pp. 215-22.

[Warren, 1977] Warren D.H.D., "Implementing Prolog — compiling predicate logic programs", Research Reports 39–40, Dept. of Artificial Intelligence, University of Edinburgh.

[Warren, 1980] Warren D.H.D., "An improved Prolog implementation which optimizes tail recursion", Research Paper 156, Dept. of Artificial Intelligence, University of Edinburgh.

[Warren, 1983] Warren, D.H.D., "An Abstract Prolog Instruction Set", Artificial Intelligence Center, SRI International, Menlo Park, CA, USA, Technical Note 309.

# 12. Appendix A—The Interpreter

This is the source code of the interpreter. A few of the definitional forms deserve more comment: defun is the normal form for defining a function. defsubst also defines a function, but the compiler is notified that calls to the function shall compile open. The variants deffun and deffsubst transform tail recursion to iteration in addition to the workings of defun and defsubst.

```
;Test for the base case of prove-conjunction before calling
;it as a function
(defsubst prove-conjunction-if-need-be (predications continuation)
  (cond ((null predications) (invoke continuation))
        (t (prove-conjunction predications continuation))))

;Prove a single predication. Used in prove-conjunction
(defsubst prove (predication interpreter-slot continuation)
  (let ((prover (interpreter-slot-prover interpreter-slot))
        (clauses (interpreter-slot-clauses interpreter-slot)))
    (cond ((null prover)
           (cond (clauses
                  (try-each clauses continuation (rest1 predication) *trail*))))
          (t (lexpr-funcall prover continuation (rest1 predication))))))

(defsubst try-first (clauses try-each-continuation arguments)
  (let ((templates (send (first clauses) ':templates)))
    (and (%unify-term-with-template arguments (first templates))
         (let ((body (%construct (rest1 templates))))
           (prove-conjunction-if-need-be body try-each-continuation)))))

;This version of try-each-clause is used in calls between
;interpreted predicates to avoid indirection through provers,
;and to help prove-conjunction tail-recurse.
(deffsubst try-each (clauses try-each-continuation arguments mark)
  (cond ((null (rest1 clauses))
         (try-first clauses try-each-continuation arguments))
        ((try-first clauses try-each-continuation arguments))
        (t (untrail mark)
           (try-each (rest1 clauses) try-each-continuation arguments mark))))


;The same as the above but calls compile closed.
(deffun try-each-interpreted-clause (clauses continuation arguments mark)
  (cond ((null clauses) nil)
        ((let ((templates (send (first clauses) ':templates)))
           (and (%unify-term-with-template arguments (first templates))
                (let ((body (%construct (rest1 templates))))
                  (prove-conjunction-if-need-be
                    body continuation)))))
        (t (untrail mark)
           (try-each-interpreted-clause
             (rest1 clauses) continuation arguments mark))))
```

```
;Discussed in Section 6.4.
(defun invoke-continuation (continuation *cut-tag*)
  (invoke continuation))

;Discussed in Chapter 6.
(deffun try-each-clause (clauses continuation arguments mark)
  (cond ((null clauses) nil)
        ((send (first clauses) ':resolve continuation arguments))
        (t (untrail mark)
           (try-each-clause (rest1 clauses) continuation arguments mark))))

;Discussed in Chapter 6.
(deffun prove-conjunction (predications continuation)
  (let* ((predication (first predications))
         (interpreter-slot
           (definition-interpreter-slot (current-definition (first predication)))))
    (cond ((interpreter-slot-deterministic interpreter-slot)
           (cond ((prove predication interpreter-slot (continuation (true)))
                  (prove-conjunction-if-need-be (rest1 predications)
                                                continuation))))
          ((null (rest1 predications))
           (prove predication interpreter-slot continuation))
          (t (let ((continuation-1
                     (continuation (funcall #'prove-conjunction
                                            (rest1 predications)
                                            continuation))))
               (prove predication interpreter-slot continuation-1))))))
```

# 13. Appendix B—Built-in predicates of LM-Prolog

Built-in predicates have their own methods for determining truth values of predications pertaining to them, and constitute the main part of LM-Prolog's runtime system. This Appendix constitutes an **incomplete** list of the built-in predicates of LM-Prolog—a **complete** list can be found in [Carlsson and Kahn, 1983].

The built-in predicates can roughly be divided into the following classes: Interface to Lisp, List Processing, Control and Meta-level, Input/Output, Arithmetic, and Database.

## 13.1 Interface to Lisp

LM-Prolog's internal representation of terms have been designed in a way that often eliminates any overhead in calling out to Lisp. But due to LM-Prolog's storage management, terms containing uninstantiated variables, and the semantics of noncanonical term the data to be passed to Lisp may need some conversion.

The Lisp interface is implemented by three built-in predicates: lisp-value, for calling a Lisp function with some arguments and getting a value back; lisp-predicate, for calling a Lisp "truth valued" function with some arguments and succeeding if it returned non-nil; and lisp-command, for telling Lisp to execute a side-effect.

They all take an optional control argument specifying what conversion, if any, is needed for the particular case. The value of the control argument can be

:copy All noncanonical terms are invoked, and the Lisp form is to be copied out of LM-Prolog's temporary area before evaluation. This could be needed in lisp-command if the function called stores its argument somewhere as a side-effect.

:invoke Noncanonical terms are invoked, but the result may still point into Prolog's temporary area. This is often used for printing.

:query Each noncanonical term encountered asks the user whether to invoke it or not. This mode of interfacing is used e.g. in the Prolog top levels—they display answers to user queries, but if the answers contain noncanonical terms, the user is queried about how they should be handled.

:dont-invoke
Lazy collections are not run, eager collections are not waited for, and constraints are not run. Arguments are passed directly to Lisp functions without conversion. In particular, uninstantiated variables will be passed as locatives. This mode is safe to use if the arguments are known to be ground, do not contain collections or constraints, and will not be stored anywhere.

## 13.2 List Processing

This subsection introduces some list processing predicates of LM-Prolog.

(APPEND *appended* . *parts*)

succeeds if *appended* is formed by appending *parts*.

(REVERSE *reversed list*)

succeeds if *reversed* is formed by reversing *list*.

(LENGTH *length list*)

succeeds if *list* is a list *length* long.

(SORT *sorted-list unsorted-list P*)

succeeds if *sorted-list* is a permutation of *unsorted-list* such that *P* imposes a total ordering over *sorted-list.*

## 13.3 Control and Meta-Level

This subsection introduces some control and meta-level primitives of LM-Prolog.

The predicates

(TRUE)
(COMMENT . *ignore*)

always succeed. The predicates

(FALSE)
(FAIL)

always fail. A conjunction is written as

(AND . *predications*)

In the body of a clause this is implicit and need not be given. A disjunction of predications is written

(OR . *predications*)

The predicate

(BAG-OF *bag term . predications*)

unifies *bag* with the list instantiations of *term* in the solutions found by exhaustively executing *predications.* Set-of is like bag-of but suppresses copies in *bag.* Both bag-of and set-of puts the instantiations in the same order as they were found in. Prefixing Set-of and bag-of with lazy- gets lazy collections. A prefix of eager- gets eager collections. These control variants of sets and bags are discussed in more detail in Chapter 8. The predicate

(PROVE-ONCE . *predications*)

finds just the first proof of *predications.* The predicate

(CAN-PROVE . *predications*)

is the "existential" variant of the above in that no bindings are made. The predicate

(CANNOT-PROVE . *predications*)

is the negative of above where negation is treated as failure to prove [Clark, 1978].

Within a clause, LM-Prolog's principal conditional construct is cases:

(CASES . *alternatives*)

where each *alternative* is a conjunction written as a list of conjuncts. This construct is equivalent to the conjunction of the elements of the first alternative whose first conjunct is provable. If there is no such alternative it is false.

There is a predicate that guarantees that predications are executed upon backtracking, mainly to ensure cleaning up of side-effects and the like:

(UNWIND-PROTECT *predication . undo-predications*) is logically equivalent to just

*predication*

but also finds all solutions to *undo-predications* upon backtracking. This is guaranteed to eventually happen, regardless of the use of control primitives, errors, or even hitting the abort button.

The *cut* ("!" or "/" in other references) is lexically scoped in LM-Prolog i.e. it will cut out all alternatives up to and including the predication that invoked the predicate containing it. It is written as

(CUT)

Predicates for inspecting the current instantiation of terms include

(IDENTICAL . *terms*)
(ATOMIC *term*)
(SYMBOL *term*)
(NUMBER *term*)
(VARIABLE *term*)
(GROUND *term*)
(FINITE *term*)

identical succeeds iff all its arguments are identical. atomic, symbol, number, and variable check the datatype of the current instantiation of the argument. ground succeeds iff its argument is fully instantiated i.e. variable-free in the run time sense. finite checks the acyclicity of its argument.

Some of these have negative counterparts by prefixing the predicator with not-.

## 13.4 Input/Output

This subsection introduces the commonest I/O predicates of LM-Prolog.

(READ *term stream*)

unifies *term* with a term read from *stream*. *Stream* is optional and defaults to the terminal. Read may be prefixed with global- to cause the term being read to be read in the same lexical environment as the top-level query that recursively invoked the global-read. I.e. global-read will recognize variable names that are identical with those occurring in the top-level query or between global reads.

(FORMAT *stream control-string* . *terms*)

This prints *terms* according to the formatting information in *control-string* onto the stream *stream*. A stream of t denotes the terminal. For details, consult [Moon *et al.*, 1983], function format.

(OPEN-FILE *stream file-name* . *options*)

This opens the file named *file-name* as a stream and unifies it with *stream*. *Options* may include :direction to indicate whether the file is being opened for :input or :output. It is guaranteed that the stream is closed upon backtracking. For details, consult [Moon *et al.*, 1983], function open.

## 13.5 Arithmetic

The arithmetical predicates include just a few very common operations. By the Lisp escape mechanism, it is extremely simple to define more.

These arithmetical predicates are "uni-directional" i.e. "input" arguments must be instantiated. More general predicates can readily be defined using *constraints*, see section 8.4.

(SUM *sum* . *addends*)
(PRODUCT *product* . *multiplicands*)

unify *sum* or *product* with the sum or product of *numbers*, respectively.

(DIFFERENCE *difference subtrahend subtractor*)
(QUOTIENT *quotient dividend divisor*)

unify *difference* or *quotient* with the difference or quotient, respectively.

($<$ . *numbers*)
($>$ . *numbers*)
($\leq$ . *numbers*)
($\geq$ . *numbers*)

are the obvious ordering predicates on numbers.

## 13.6 Database

The principle means of creating, redefining, and modifying predicates is through the use of define-predicate. Its syntax is one of

(DEFINE-PREDICATE *predicator* (:options . *options*) . *clauses*)
(DEFINE-PREDICATE *predicator* . *clauses*)

Each clause in *clauses* must have a head whose first element is *predicator*. Each option in *options* is a list beginning with an option keyword. The following options are relevant:

(:type *type*)
: where *type* is :static (default) or :dynamic, declares whether clauses may subsequently be added to or removed from the definition;

(:world *world*)
: where *world* defaults to :user, says what world the predicate is to be added to;

(:if-another-definition *action*)
: controls what happens if a predicate is defined in more than one world in the universe. At run time, the other definition will be tried before (*action* = :try − before) this one, after (*action* = :try − after) this one, or be :ignored (the default);

(:if-old-definition *action*)
: controls what happens if there is already a predicate defined in the same world as this one. If *action* is :flush (the default), the new one supersedes the old. If it is :keep, the new clauses are added to the old predicate;

(:place-clauses *where*)
: controls, when adding new clauses to a predicate, where those clauses go. They can go :before or :after the other ones, or :anywhere, letting LM-Prolog decide;

(:indexing-patterns . *patterns*)

adds indexing to the search mechanism for matching clauses. Each *pattern* is a *search pattern* and will add one index. A search pattern is a cons of *predicator* and an argument pattern. In the argument pattern there should be exactly one symbol beginning with a '+' specifying what the key is for that particular index. At run time, the first index whose key is bound will be used.

(:deterministic *when*)

This option may be used to force a predicate to have at most one solution (*when* = :always). The default is to not be deterministic. The compiler often determines on its own that a predicate is deterministic, and if so, emits this declaration to the benefit of subsequent compilations of predications.

(:execution *mode*)

This option controls whether a definition is :interpreted or :compiled.

(:compile-method *method*)

This is used for compiled predicates for controlling the subsequent compilation of **predications** pertaining to it. The default is :closed, ordinary out-of-line compilation. *Method* = :open causes in-line compilation of predications. *Method* = :intrinsic is for code generation of built-in primitives.

Let us consider some examples of definitions:

```
(define-predicate a
  (:options (:type :dynamic) (:world :jupiter))
  ...)
```

This defines a predicate a which is assertable. It is added to the world :jupiter. If there is an old predicate a in :jupiter, the old one flushed first.

```
(define-predicate b
  (:options (:deterministic :always) (:if-another-definition :try-before))
  ...)
```

This defines a non-assertable predicate b which is added to the default world (initially :user). Now, assume that the universe is (:user :jupiter :system) at run time. Assume also that there is another predicate b in :jupiter. If b is called, the semantics is like appending the clauses of b in :jupiter in front of those of b in :user. Moreover, b is forced to become deterministic.

```
(define-predicate c
  (:options (:if-old-definition :keep) (:place-clauses :after))
  ...)
```

Some clauses are added to the predicate c, which must be dynamic. The new clauses are appended after the old ones.

Other database predicates include

(ASSERT *clause world*)
(ASSERT *clause*)

for adding a clause to a world,

(RETRACT *clause world*)
(RETRACT *clause*)

for removing matching clauses,

(ADD-WORLD *world*)

for adding a world to the universe,

(REMOVE-WORLD *world*)

for removing a world from the universe,

(PRINT-DEFINITION *predicator*)

for viewing a definition.

# 14. Appendix C—Sample compilations

This chapter contains samples of compiled LM-Prolog predicates. The are taken from a list of benchmarks included in [Warren, 1977]. See [Moon *et al.*, 1983] for details about the Lisp Machine instruction set.

## 14.1 Concatenate

```
(DEFINE-PREDICATE CONCATENATE
  (:OPTIONS (:WORLD :BENCHMARKS) (:EXECUTION :COMPILED) (:ARGUMENT-LIST (+ + -)))
  ((CONCATENATE (?FIRST . ?REST) ?BACK (?FIRST . ?ALL-BUT-FIRST))
   (CONCATENATE ?REST ?BACK ?ALL-BUT-FIRST))
  ((CONCATENATE () ?BACK ?BACK)))
```

Mapping of variables:

```
; ?FIRST → (CAR G0634), source term 202
; ?REST → (CDR G0634)
; ?BACK → ?VARIABLE1
; ?ALL-BUT-FIRST → ?VARIABLE5, source term 200
```

Compiled code:

```
(DEFFUN CONCATENATE-IN-BENCHMARKS-PROVER
        (.CONTINUATION. ?VARIABLE0 ?VARIABLE1 ?VARIABLE2 &AUX ?VARIABLE5)
  (LET ((G0364 (%DEREFERENCE ?VARIABLE0)))
    (COND ((AND (AND (CONSP G0364) T T))
           (AND (%UNIFY-TERM-WITH-TEMPLATE ?VARIABLE2
                                           (LET ((G0365 (CAR G0364))) '(3 (2 . 202) 1 . 200)))
                (FUNCALL 'CONCATENATE-IN-BENCHMARKS-PROVER
                         (CONTINUATION (INVOKE .CONTINUATION.))
                         (CDR G0364) ?VARIABLE1 ?VARIABLE5)))
          ((AND (EQ G0364 NIL))
           (AND (UNIFY (%DEREFERENCE ?VARIABLE1) (%DEREFERENCE ?VARIABLE2))
                (INVOKE .CONTINUATION.))))))
```

which further compiles to

```
20 DEREFERENCE D-PDL ARG|1 ;?VARIABLE0
21 POP LOCAL|1
22 BR-ATOM 37
23 MOVE D-PDL ARG|3 ;?VARIABLE2
24 CAR D-PDL LOCAL|1
25 POP LOCAL|2
26 MOVE D-PDL FEF|6 ;'(3 (2 . 202) 1 . 200)
27 (MISC) %UNIFY-TERM-WITH-TEMPLATE D-PDL
30 BR-NIL-POP 35
31 CDR D-PDL LOCAL|1
32 MOVE D-PDL LOCAL|0 ;?VARIABLE5
33 POP ARG|3 ;?VARIABLE2
34 BR 21
35 MOVE D-RETURN PDL-POP
36 MOVE D-IGNORE LOCAL|1
37 BR-NOT-NIL 47
40 DEREFERENCE D-PDL ARG|2 ;?VARIABLE1
41 DEREFERENCE D-PDL ARG|3 ;?VARIABLE2
42 (MISC) %UNIFY-TERM-WITH-TERM D-PDL
43 BR-NIL-POP 46
44 MOVE D-PDL ARG|0 ;.CONTINUATION.
45 (MISC) %INVOKE D-RETURN
46 MOVE D-RETURN PDL-POP
47 (MISC) FALSE D-RETURN
```

Instruction 26 references the source term for (?first . ?all-but-first).

Tail recursion was transformed to iteration (instructions 31–34). In performing that transformation, the compiler noted that dereferencing ?VARIABLE0 is only needed initially since CDR (instruction 31) subsumes dereferencing.

Note the efficient compilation of the matching of ?VARIABLE0 (instructions 22 and 37).

## 14.2 Naïve-reverse

```
(DEFINE-PREDICATE NAIVE-REVERSE
  (:OPTIONS (:WORLD :BENCHMARKS) (:EXECUTION :COMPILED) (:ARGUMENT-LIST (+ -)))
  ((NAIVE-REVERSE (?FIRST . ?REST) ?REVERSE)
   (NAIVE-REVERSE ?REST ?REST-REVERSED)
   (CONCATENATE ?REST-REVERSED (?FIRST) ?REVERSE))
  ((NAIVE-REVERSE () ())))
```

Mapping of variables:

```
; ?FIRST → (CAR G0366)
; ?REST → (CDR G0366)
; ?REVERSE → ?VARIABLE1
; ?REST-REVERSED → ?VARIABLE4
```

Compiled code:

```
(DEFFUN NAIVE-REVERSE-IN-BENCHMARKS-PROVER
        (.CONTINUATION. ?VARIABLE0 ?VARIABLE1 &AUX ?VARIABLE4)
  (LET ((G0366 (%DEREFERENCE ?VARIABLE0)))
    (COND ((AND (AND (CONSP G0366) T T))
           (AND (PROGN (SETQ ?VARIABLE4 (%CELL0))
                       (AND (FUNCALL 'NAIVE-REVERSE-IN-BENCHMARKS-PROVER
                                     (CONTINUATION (TRUE)) (CDR G0366) ?VARIABLE4)
                            (FUNCALL (CURRENT-ENTRYPOINT 'CONCATENATE)
                                     (CONTINUATION (INVOKE .CONTINUATION.))
                                     ?VARIABLE4 (PROLOG-LIST (CAR G0366)) ?VARIABLE1)))))
          ((AND (EQ G0366 NIL))
           (AND (UNIFY (%DEREFERENCE ?VARIABLE1) NIL) (INVOKE .CONTINUATION.))))))
```

which further compiles to:

```
30 DEREFERENCE D-PDL ARG|1 ;?VARIABLE0
31 POP LOCAL|1
32 BR-ATOM 57
33 (MISC) %CELL0 D-PDL
34 POP LOCAL|0 ;?VARIABLE4
35 CALL D-PDL FEF|7 ;'NAIVE-REVERSE-IN-BENCHMARKS-PROVER
36 MOVE D-PDL FEF|8 ;'(TRUE)
37 CDR D-PDL LOCAL|1
40 MOVE D-LAST LOCAL|0 ;?VARIABLE4
41 BR-NIL-POP 55
42 MOVE D-PDL FEF|9 ;'CONCATENATE
43 MOVE D-PDL FEF|10 ;'#<DTP-LOCATIVE 14304536>
44 (MISC) %CURRENT-ENTRYPOINT D-PDL
45 CALL D-RETURN PDL-POP
46 MOVE D-PDL ARG|0 ;.CONTINUATION.
47 MOVE D-PDL LOCAL|0 ;?VARIABLE4
50 CAR D-PDL LOCAL|1
51 REFERENCE D-PDL PDL-POP
52 MOVE D-PDL '1
53 (MISC) %PROLOG-LIST D-PDL
54 MOVE D-LAST ARG|2 ;?VARIABLE1
55 MOVE D-RETURN PDL-POP
56 MOVE D-IGNORE LOCAL|1
57 BR-NOT-NIL 67
60 DEREFERENCE D-PDL ARG|2 ;?VARIABLE1
61 MOVE D-PDL 'NIL
62 (MISC) %UNIFY-TERM-WITH-TERM D-PDL
63 BR-NIL-POP 66
64 MOVE D-PDL ARG|0 ;.CONTINUATION.
65 (MISC) %INVOKE D-RETURN
66 MOVE D-RETURN PDL-POP
67 (MISC) FALSE D-RETURN
```

Tail recursion optimization is not applicable here. One variable has to be initialized by an explicitly allocated value cell (instructions 33–34). Instructions 42–44 show an example of the %CURRENT-ENTRYPOINT instruction. The locative in instruction 43 is a pointer to the definitions of CONCATENATE. The pointer is installed at load time.

In the call to CONCATENATE, a list 1 long is constructed by our %PROLOG-LIST instruction (instructions 50–53).

## 14.3 Partition

```
(DEFINE-PREDICATE PARTITION
   (:OPTIONS (:WORLD :BENCHMARKS) (:EXECUTION :COMPILED) (:ARGUMENT-LIST (+ + - -)))
 ((PARTITION (?X . ?L) ?Y (?X . ?L1) ?L2)
  (> ?Y ?X)
  (CUT)
  (PARTITION ?L ?Y ?L1 ?L2))
 ((PARTITION (?X . ?L) ?Y ?L1 (?X . ?L2))
  (PARTITION ?L ?Y ?L1 ?L2))
 ((PARTITION () ? () ())))
```

Mapping of variables:

```
; ?X → (CAR G0367), source term 203
; ?L → (CDR G0367)
; ?Y → ?VARIABLE1
; ?L1 → ?VARIABLE6 and source term 200 in clause 1, ?VARIABLE2 in clause 2.
; ?L2 → ?VARIABLE3 in clause 1, ?VARIABLE6 and source term 200 in clause 2.
```

Compiled code:

```
(DEFFUN PARTITION-IN-BENCHMARKS-PROVER
        (.CONTINUATION. ?VARIABLE0 ?VARIABLE1 ?VARIABLE2 ?VARIABLE3 &AUX ?VARIABLE6)
  (LET ((G0367 (%DEREFERENCE ?VARIABLE0)))
    (LET ((MARK *TRAIL*))
      (COND ((AND (AND (CONSP G0367) T T)
                  (%UNIFY-TERM-WITH-TEMPLATE ?VARIABLE2
                                             (LET ((G0368 (CAR G0367)))
                                               '(3 (2 . 203) 1 . 200)))
                  (COND (NIL)
                        ((AND (AND (LEXPR-FUNCALL #'>
                                                  (%DEREFERENCE ?VARIABLE1)
                                                  (PROLOG-LIST (CAR G0367)))
                                   (INVOKE (CONTINUATION (TRUE))))))))
             (COND ((FUNCALL 'PARTITION-IN-BENCHMARKS-PROVER
                             (CONTINUATION (INVOKE .CONTINUATION.))
                             (CDR G0367) ?VARIABLE1 ?VARIABLE6 ?VARIABLE3))))
            ((PROGN (UNTRAIL MARK) NIL))
            ((AND (AND (CONSP G0367) T T))
             (AND (%UNIFY-TERM-WITH-TEMPLATE ?VARIABLE3
                                             (LET ((G0369 (CAR G0367)))
                                               '(3 (2 . 203) 1 . 200)))
                  (FUNCALL 'PARTITION-IN-BENCHMARKS-PROVER
                           (CONTINUATION (INVOKE .CONTINUATION.))
                           (CDR G0367) ?VARIABLE1 ?VARIABLE2 ?VARIABLE6)))
            ((AND (EQ G0367 NIL))
             (AND (UNIFY (%DEREFERENCE ?VARIABLE2) NIL)
                  (UNIFY (%DEREFERENCE ?VARIABLE3) NIL)
                  (INVOKE .CONTINUATION.)))))))
```

The Lisp function further compiles to

```
22 DEREFERENCE D-PDL ARG|1 ;?VARIABLE0
23 POP LOCAL|1
24 MOVE D-PDL FEF|6 ;*TRAIL*
25 POP LOCAL|2 ;MARK
26 MOVE D-IGNORE LOCAL|1
27 BR-ATOM 47
30 MOVE D-PDL ARG|3 ;?VARIABLE2
31 CAR D-PDL LOCAL|1
32 POP LOCAL|3
33 MOVE D-PDL FEF|7 ;'(3 (2 . 203) 1 . 200)
34 (MISC) %UNIFY-TERM-WITH-TEMPLATE D-IGNORE
35 BR-NIL 47
36 DEREFERENCE D-PDL ARG|2 ;?VARIABLE1
37 CAR D-PDL LOCAL|1
40 DEREFERENCE D-PDL PDL-POP
41 > PDL-POP
42 BR-NIL 47
43 CDR D-PDL LOCAL|1
44 MOVE D-PDL LOCAL|0 ;?VARIABLE6
45 POP ARG|3 ;?VARIABLE2
46 BR 23
47 MOVE D-PDL LOCAL|2 ;MARK
50 (MISC) %UNTRAIL D-IGNORE
51 MOVE D-IGNORE LOCAL|1
52 BR-ATOM 67
53 MOVE D-PDL ARG|4 ;?VARIABLE3
54 CAR D-PDL LOCAL|1
55 POP LOCAL|3
56 MOVE D-PDL FEF|7 ;'(3 (2 . 203) 1 . 200)
57 (MISC) %UNIFY-TERM-WITH-TEMPLATE D-PDL
60 BR-NIL-POP 65
61 CDR D-PDL LOCAL|1
62 MOVE D-PDL LOCAL|0 ;?VARIABLE6
63 POP ARG|4 ;?VARIABLE3
64 BR 23
65 MOVE D-RETURN PDL-POP
66 MOVE D-IGNORE LOCAL|1
67 BR-NOT-NIL 103
70 DEREFERENCE D-PDL ARG|3 ;?VARIABLE2
71 MOVE D-PDL 'NIL
72 (MISC) %UNIFY-TERM-WITH-TERM D-PDL
73 BR-NIL-POP 102
74 DEREFERENCE D-PDL ARG|4 ;?VARIABLE3
75 MOVE D-PDL 'NIL
76 (MISC) %UNIFY-TERM-WITH-TERM D-PDL
77 BR-NIL-POP 102
100 MOVE D-PDL ARG|0 ;.CONTINUATION.
101 (MISC) %INVOKE D-RETURN
102 MOVE D-RETURN PDL-POP
103 (MISC) FALSE D-RETURN
104 MOVE D-RETURN PDL-POP
```

The compiler is not clever enough to notice that the pattern for the first argument is the same in the first two clauses. It duplicates the test (instructions 26–27 and 51–52).

Instructions 33 and 56 reference the source term for (?x . ?l1) and (?x . ?l2).

Tail recursion optimization was applicable (instructions 43–46, 61–64).

Instructions 36–41 illustrates the compilation of e.g. arithmetic tests. The predicate > is defined as

```
(define-predicate >
  (:options (:compile-method :open))
  ((>))
  ((> ?x . ?arguments) (lisp-predicate (lexpr-funcall #'> '?x '?arguments))))
```

and so the predication

```
(> ?Y ?X)
```

expands into

```
(or (fail);;since the first head doesn't match
   (lisp-predicate (lexpr-funcall #'> '?y '(?x))))
```

which corresponds to the following segment of the Lisp function:

```
(COND (NIL)
      ((AND (AND (LEXPR-FUNCALL #'>
                                (%DEREFERENCE ?VARIABLE1)
                                (PROLOG-LIST (CAR G0367)))
                 (INVOKE (CONTINUATION (TRUE))))))
```

Various *compiler optimizers* transforms the code segment to:

```
(> (%DEREFERENCE ?VARIABLE1) (CAR G0637))
```

which compiles to instructions 36–41.

# 15. Appendix D—Instruction set extensions

The CADR Machine features writeable control store and moreover has 38 unused opcodes and more than 4000 unused control memory locations.

This has been exploited in the implementation by adding 12 new instructions occupying 383 words of control store, 27 dispatch memory locations and 3 "A" memory words. A comparison with a version of LM-Prolog that does not use an extended instruction set suggest a factor of 3 speed difference. Moreover, the version without microcode support has to emit much more in-line code in order to not further deteriorate performance of compiled code.

(%untrail mark)
: This unwinds the trail stack until its stack-pointer is mark. Value cells are made unbound and continuations are invoked. Occurrences of this instruction correspond to "backtrack points". It is typically emitted between code for subsequent clauses in compiled code.

(%unify-term-with-term x y)
: This unifies the object terms x and y. It is used in compiled code and by all parts of the runtime system that want to unify object terms.

(%unify-term-with-template x y)
: This unifies the object term x with the source term y. There is typically one occurrence of this instruction for each non-variable argument of each clause of a compiled predicate. In the interpreter, this instruction is applied to source terms for heads of clauses.

(%construct source-term)
: This constructs a term out of source-term. The principal occurrence of this instruction is in the interpreter, which uses it to construct instances of bodies of clauses. The compiler emits inline code for constructing instances of arguments of predications instead.

(%cell name)
: This makes a named value cell two words long. Named value cells are created by the interpreter. The instruction occurs at a few places in the runtime system.

(%cell0)
: This makes an unnamed value cell one word long. Unnamed value cells are created by compiled code. The instruction is emitted when a void variable is an argument of a predication inside a compiled predicate.

(%reference term)
: This turns term into a forwarding pointer if it is a value cell. Its main use is in inline code for constructing instances of arguments of predications. It also occurs at a few places in the runtime system.

(%dereference term)
: This dereferences term if it is a value cell. The rationale for having it is that when unbound variables are first passed as arguments, then bound, and then accessed, there is no automatic following of forwarding pointers, since the accesses

don't go via car or cdr. %dereference does it instead. Previously, %dereference was commonly used together with %unify-term-with-template but the latter instruction now dereferences its first argument, thus saving many instructions from being emitted. The main use of %dereference is now in interfacing to Lisp functions.

(%prolog-list element... length)

This instruction substitutes for the Lisp function list-in-area, which needs full-fledged function calls, for memory allocation in *prolog-work-area*. The motivation for this instruction is to avoid the overhead of function calls when constructing terms.

(%prolog-list* element... length)

This instruction substitutes for the Lisp function list*-in-area. Analogous to the above.

(%invoke continuation)

This invokes continuation (currently, applies its car to its cdr) after checking for two common cases first. If the function is true, a common case due to determinacy optimizations, or if it is false, a common case due to sets and bags, t or nil is returned, respectively. Otherwise, a function call takes place. Again, the motivation is to avoid function calls, however the motivation is somewhat weak here since calls to true and false are fast anyway. But then this instruction takes one macro-instruction less than the obvious sequence using apply.

(%current-entrypoint predicator defs-location)

This implements the cached search for predicator's prover. Defs-location is a locative to predicator's definition alist. The instruction optimizes the most common case where there is a hit. Without it, there would be 10 more instructions of inline code associated with each predicate-to-predicate call, for covering the hit and miss cases.

The occur check is a microcoded function, not invoked as an instruction though.

The microcode has direct access to the special variables *trail-array*, *prolog-work-area*, and *vector*. "Direct access" means that the variables reside in internal processor memory "A" so that the microcode can get at their values without virtual memory read cycles. The macrocode consequences are (*i*) these variables cannot be closure variables, and (*ii*) accessing these variables becomes a trifle slower for the part of the runtime system that is not microcoded.

Using the ordinary virtual memory mechanism, the microcode also has access to the variable *universe*, to the function array-push-extend, to some of its own dtp-u-entries which it needs when it runs out of microcode subroutine stack and so has to recurse "slowly", and finally to the definition of read-only-variable.