Open Prolog: a Structure-Sharing Prolog for the Macintosh

A thesis submitted to the
University of Dublin, Trinity College,
for the degree of
Doctor of Philosophy

Michael Brady
August 2005

Declaration

I, the undersigned, declare that this work has not previously been submitted as an exercise for a degree at this, or any other University, and that unless otherwise stated, is my own work.

_____ August 7, 2005

Michael Brady

Permission to Lend or Copy

I, the undersigned, agree that Trinity College Library may lend or copy this thesis upon request.

_____ August 7, 2005

Michael Brady

Acknowledgements

It is a pleasure to acknowledge the help, support and guidance of my supervisor, Professor John G. Byrne. I should also like to thank David M. Abrahamson and Pádraig Cunningham for their encouragement. Thanks are also due to John Gallagher, formerly of Trinity College, for getting me interested in Prolog in the first place.

Mike Brady, Dublin, August 2005.

Contents

List of Figures

List of Tables

Abstract

This thesis explores the design and implementation of a Prolog system with just one mode of program execution rather than the two modes of execution—interpretation and compilation—present in most Prolog implementations in use today.

The main contribution of the thesis is such a design, combining reasonably high runtime performance with the ability to modify, debug and inspect program code. The design has been realised as *Open Prolog*, a complete Prolog environment with an integrated editor and debugger, built for the Apple Macintosh with a Motorola 68000 instruction set.

The implementation supports the argument that the design offers high speed compilation combined with reasonably high speed implementation and simplicity of implementation, and compares favourably, in terms of speed of execution and ease of program development, with a number of other implementations.

This work is concerned with the design and implementation of Open Prolog: the overall architecture, static and dynamic aspects of the system, including code representation, the image machine, compilation, data areas, the runtime behaviour of the system, built-in predicates and garbage collection. The performance of the implementation is benchmarked and compared with a number of other implementations, and the results of a low-level profiling study are presented.

Contribution

The principal contribution is a design for a Prolog implementation based on relatively simple principles that combines high speed compilation and reasonable runtime performance with the ability to modify, inspect and debug Prolog code.

Influenced by principles of Direct Correspondence Architectures, the design is based on an abstract machine with an instruction set such that every aspect of the source code is represented in the machine code. This facilitates the decompilation of machine code, i.e. the reconstruction of the source code, for inspection and debugging purposes. It also simplifies compilation.

A number of subsidiary contributions are made with the intention of enhancing the design in various ways. These and suggestions for further development of the implementation are made at the end of the work.

Chapter 1

Introduction

There are two main approaches to efficient Prolog implementation: emulated code and native code. Emulated code compiles to an abstract machine and is interpreted at run-time. Native code compiles to the target machine and is executed directly. [71, p5]

Most Prolog compilers, whether they target an abstract machine or native code, do not permit inspection or modification of the compiled code. Compilation usually destroys any obvious correspondence between the source code and the executable code, and the compilation process can take a considerable time.

While a program is being developed, outright runtime performance is often less important that the ability to examine and modify the program, and it is useful [43, p9] to be able to modify programs even in completed fully-debugged programs. Compiler-based Prolog implementations typically offer an interpreted mode of execution to facilitate debugging and to support the dynamic assertion and retraction of clauses. Programs that are interpreted on these systems typically run an order of magnitude slower than if they were compiled; memory requirements may also be high. To speed up the storage and retrieval of interpreted clauses, interpreters generally perform fast simple

transformations of source code to interpreted representation, for instance in Quintus Prolog, the interpreted representation of clauses is designed so that an instance of the clause can be created on the global stack as quickly as possible [20]. So, whereas interpreted programs may execute more slowly, interpreters generally offer faster reconsult times, fewer runtime restrictions, and better inspection and debugging facilities than compilers.

A question arises as to whether the advantages of compilation and interpretation could be combined in an implementation with just one mode of operation. That is, could an implementation be built with one uniform mode of operation which was fast and memory-efficient and which also offered facilities for code inspection, modification and debugging?

A single mode of operation is attractive because it has the potential to simplify both the design and implementation of a complete system. For example, instead of requiring one representation for compiled clauses and another for interpreted clauses, a single representation would be conceptually simpler and could lead to a simpler implementation. As a second example, arrangements would have to be made to enable compiled code to call interpreted code and *vice versa*; this would be unnecessary with a single mode of operation. There is also, it is ventured, an aesthetic reason for preferring the conceptually simpler method of implementation. Gelerntner [34] argues for the combination of simplicity and power as the essence of what he calls *machine beauty*. Certainly the idea that one mode of operation might supplant two was attractive to this writer.

The aim of this thesis was to design just such a Prolog implementation; that is, having one mode of operation that would combine the advantages of compiler-based implementations—speed and memory efficiency—with those of interpreter-based implementations, which include ease of inspection, ease of modification and ease of debugging of Prolog programs.

To arrive at this design and the subsequent implementation, the following

research questions were addressed:

- Broadly speaking, how should a single mode of operation be implemented? Should it be based on compilers, interpreters, or some combination?

- How should programs, data, data structures and, importantly, dynamic data structures be represented? For dynamic data structures, two possibilities are *structure sharing* [9] and *structure copying* [13, 51].[1]

- In view of the requirement that programs be viewable and modifiable as well as executable—and allowing that Prolog operates in the main by performing unification—what is the best way to organise program execution? For instance, should the program be converted into host machine code or into code for an abstract machine that in turn is emulated? If the latter, how should the abstract machine be organised?

- How should the system deal with modifiable code? How does this interact with memory management and clause selection techniques such as indexing?

- How can Prolog's control structures, such as disjunction or catch-and-throw error handling be implemented efficiently?

- How does the resulting implementation compare with others?

- What performance bottlenecks can be identified?

These questions, of course, are inter-related. For instance, the program representation scheme and the architecture of the abstract machine, if there is one, are intimately connected.

[1] These techniques are discussed in Section 1.5.1 on page 19.

1.0.1 Plan of the Thesis

The rest of this chapter introduces logic programming and Prolog, and deals briefly with the early history of Prolog implementations. Chapters 2 and 3 examine arguably the two most influential Prolog implementation architectures, the DEC10-Prolog Machine—the 'PLM'—which is based on structure sharing, and the Warren Abstract Machine—the 'WAM', which is based on structure copying.

Chapter 4 considers the design issues and presents an overview of the architecture of Open Prolog and this is followed by four short chapters on aspects of the implementation: *compilation*, treatment of *built in predicates*, *control constructs & exception handling* and *memory management*.

Chapter 9 contains some benchmarking and profiling results and Chapter 10 summarises the work and presents some suggestions for further work.

1.1 Prolog and Logic Programming

The thing that distinguishes logic programming from other kinds of programming is that executing a logic program is the same as proving a theorem. Kowalski [45] characterises logic programming as:

> *"... the use of logic to represent knowledge and the use of deduction to solve problems by deriving logical consequences.*
> ...
> *It exploits the fact that logic can be used to express definitions of computable functions and procedures; and it exploits the use of proof procedures that perform deductions in a goal-directed manner, to run such definitions as programs."*

This 'liberal' [45] view of logic programming admits of a variety of formalisms and deduction mechanisms, and there are many logic programming languages. Prolog is perhaps the best known logic programing language. Other notable logic programming languages include, for instance, general

purpose logic progamming languages Prolog III [24], Gödel [36] and Mercury [62]; constraint-handling logic programming systems CLP(R) [40] and parallel-execution languages Concurrent Prolog [61] and PARLOG [35].

In Prolog [23], the formalism is based on Horn Clauses, and the deduction system is *SLD Resolution* [46]. Developed by Alain Colmerauer and Philippe Roussel [58] at the Université Aix-Marsellies[2], the name 'Prolog' was suggested by Phillipe Roussel's wife Jacqueline, as an abbreviation for *pro*grammation en *log*ique [25]. Prolog has been standardised by the International Standards Organisation [39, 28]. See also the on-line entry for Prolog on the Free Online Dictionary of Computing [80].

A Prolog program[3] consists of an ordered set of axioms, represented in the form of *clauses*, along with the *goal*, which is a query or conjecture the system attempts to prove using the axioms provided. If successfully proven, the query, along with any values substituted for variables in the query, is a *theorem*. Hence, proving a query is the same as deriving a theorem. Figure 1.1 illustrates the makeup of a Prolog program. Clauses generally have two main components—a *head* which contains exactly one term—the *head term*—and a *body* which may be empty. The axiom represented by a clause is that the predicate represented by the head term is true if the predicates represented by the body are true. The predicate represented by a clause is identified by the principal functor of the head term. Thus, for example, the clauses:

```
cat(pangur).
animal(X) :- cat(X).
```

are two axioms, each terminated by a full stop. The principal functor of the head term of the first clause has the name `cat` and an arity of `1`; the

[2]See [25], [45] and [23] for interesting accounts of the origins of the language, from which much of the material in this section is derived.

[3]For a comprehensive introduction to the language, see [22] or [11].

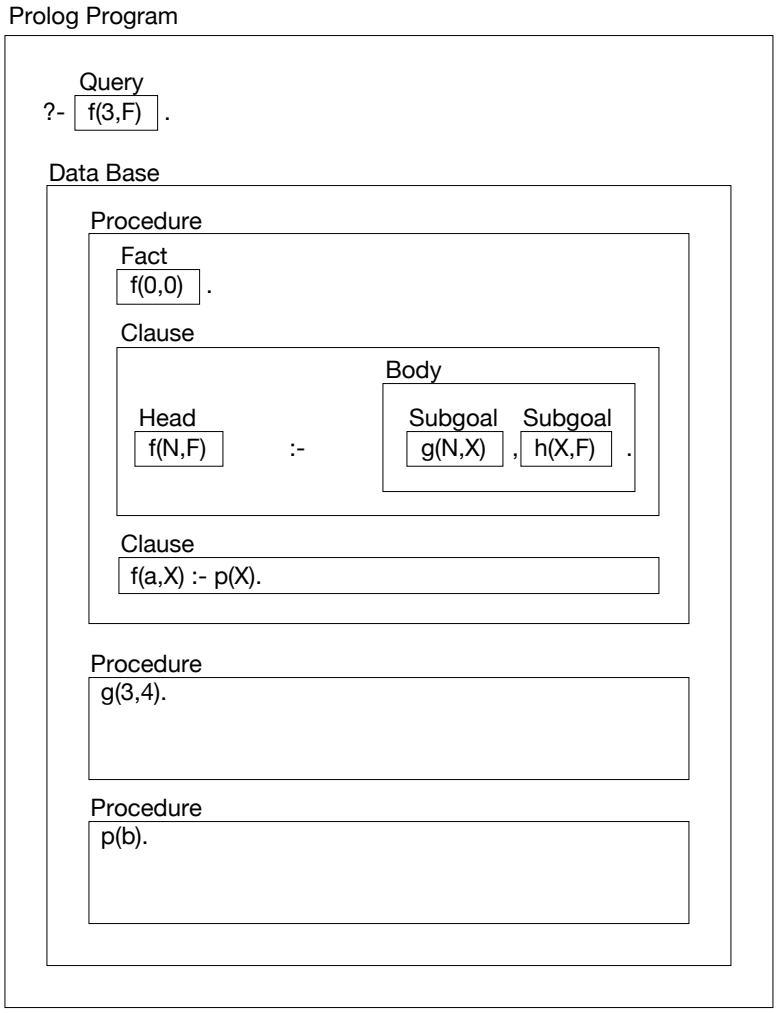

Figure 1.1: The layout of a Prolog program. A program consists of a query to be proved, a *conjecture*, and an ordered set of axioms, represented as Prolog *clauses*. Proving the conjecture is the *goal* of the Prolog execution mechanism and proving each conjecture it depends on becomes, recursively, a new *subgoal* of the execution mechanism. In a similar vein, an ordered set of clauses with the same head term is called a *procedure*. The diagram is taken from [31].

clause is an axiom stating that the predicate `cat/1` is true in respect of the argument `pangur`.[4] Similarly, the second clause is an axiom stating that the predicate `animal/1` is true in respect of some argument denoted by the universally quantified variable `X`.[5] Clauses with the same head term represent alternative ways of proving the same predicate and are called, collectively, a *procedure*.

The body of a clause lists the predicates that must be true for the predicate denoted by the head term to be true in respect of its arguments. If the body is empty, the predicate is unconditionally true of its arguments, as it depends on no other predicates, and the clause is called a *fact*. For example, the first clause is a fact that states that `cat/1` is always true in respect of its argument `pangur`.

If the body of a clause is non-empty, the clause is called a *rule* and the body represents the predicates that, when true with the given arguments, imply that the predicate represented by the clause's head term is true in respect of its arguments. The second clause states that `animal/1` is true in respect of its argument `X` whenever the predicate `cat/1` is true in respect of the same argument.

Kowalski goes on to spell out the significance of using logic:

> *A consequence of using logic to represent knowledge is that such knowledge can be understood* declaratively*. A consequence of using deduction to derive consequences in a computational manner is that the same knowledge can also be viewed* procedurally *Thus, logic programming allows us to view the same knowledge both declaratively* and *procedurally." [45]*

In Prolog, the knowledge is represented using Horn clauses, and the deduction mechanism is *SLD resolution* [46]; this mechanism is responsible

[4]The name `pangur` identifies a simple named constant called an *atom*. Numbers are also allowed as constants in Prolog.

[5]Variables in Prolog are denoted by identifiers starting with a capital letter or an underscore. The scope of a variable is the clause in which it appears.

for Prolog's goal-directed theorem-proving behaviour. Thus, apart from the static declarative logical interpretation—*"this predicate is true if the predicates in the clause body are true"*—Prolog clauses have a prescriptive interpretation—*"to prove the predicate, prove all the predicates in the clause body"*, and this is the basis for executing Prolog programs.

Formally, Prolog works by refutation: that is, to prove a conjecture, the conjecture is assumed to be false and, taking the negated conjecture and all the clauses in the program, a sequence of logical steps is sought that leads to a contradiction. The contradiction refutes the falsity of the conjecture, and thus proves the conjecture under the circumstances discovered, which may include assigning particular values to variables in the conjecture. Each logical step in the proof of the contradiction is made using *resolution*, a technique developed by Alan J. Robinson [57].

Here is a statement of the rule of resolution (after [63, p62]): from the two clauses: P :- $Q_1, ..., Q_j, ..., Q_n$. and Q :- $R_1, ..., R_m$., if there exists a *substitution* s such that $Q[s] = Q_j[s]$, a new clause can be derived: P :- $Q_1, ..., Q_{j-1}, R_1, ..., R_m, Q_{j+1}, ..., Q_n$. This new clause is the *resolvent* of the two clause on Q_j under the substitution s, obtained by replacing the literal Q_j in the first clause by the whole body of the second clause and by applying the substitution s to the whole resulting clause. The substitution s is obtained by *unification* of Q_j with a private 'copy' of the second clause.

Unification operates as follows: two simple terms can be unified if they are identical. If one term is a variable, it can be made equal to the other; the terms thus become identical and can be unified. Similarly, if both terms are variables, their identities can be merged so that they become one variable henceforward and the two terms become identical and can be unified. If none of the above steps are applicable, unification fails. Compound terms (i.e. terms containing arguments) are unified by ensuring the principal functors are identical and by recursively unifying each corresponding argument.

(A further element of unification is that an element can not be unified with another element of which it is a component. Testing for this condition—the *occur check*—is not performed in Prolog.)

The successful application of the rule of resolution above means that the newly formed clause—the resolvent—is a logical consequence of the two clauses. *Linear* resolution, where every resolution step involves a clause taken from the program and the goal that was produced in the previous resolution step, permits a more systematic method of searching for a refutation. The further restriction of selecting the goals within a literal in a fixed order, yielding *Selected-literal Linear resolution for Definite clauses (SLD resolution)*, is the basis of the Prolog execution mechanism.

SLD resolution can also be seen as a pattern-matching process: the execution mechanism searches through the clauses in the program for a clause with a head term that might be unifiable with the conjecture. On finding such a clause, a distinct copy of the clause, complete with its own private variables, is *instantiated*, or brought into existence. An attempt is made to unify the conjecture with the clause instance's head term. If the unification is unsuccessful, the execution mechanism discards the clauses instance and searches for another clause. However, if the unification is successful, the logical inference is made that the conjecture is provable using the axiom embodied in the clause instance. If the clause instance is a fact, i.e. if it has no body, then the inference is unconditional. For example, if the conjecture is `cat(X)` (i.e. that the predicate `cat/1` is true for some variable `X`), then the clause instance `cat(pangur)` may be chosen to prove the conjecture, where the conjecture's variable `X` is unified with the constant `pangur`. Since `cat(pangur)` is a fact, the conjecture `cat(X)`, where `X` has the value `pangur`, is proved unconditionally by this clause instance.

If a clause instance is a rule, then the inference made by resolving the conjecture with the clause instance is conditional on the predicates in the

body of the clause instance being true. For example, if the conjecture is `animal(T)`, (i.e. that the predicate `animal/1` is true for some argument `T`), then the clause instance `animal(X) :- cat(X)` may be chosen, where the instance's variable `X` is unified with the conjecture's argument `T`, giving a complete clause instance `animal(T) :- cat(T)`. Since this is a rule, the truth of the conjecture `animal(T)` is conditional on the predicate `cat(T)` being true. This can be proved using an instance of the first clause, as outlined in the previous paragraph, unifying the conjecture's argument `T` with the clause instance's argument `pangur`.

As described in the foregoing paragraphs, a Prolog system allows a programmer to describe a system in terms of things that are true about it—that is, the facts and rules that capture the essential properties of the system. To prove something about the system, a conjecture is made which the Prolog execution mechanism attempts to prove, using SLD resolution to make logical inferences from the axioms supplied and stored as clauses, producing a result which is a theorem, and which does not depend for its veracity on any details of the execution mechanism.

The efficiency with which a theorem is proved can be influenced by clause and goal ordering. Although the order in which the predicates are listed in the body of clause is unimportant from a logical point of view, (since all predicates in the clause body must be true for the predicate to be true), and although the order of the clauses in a procedure is also unimportant from a logical point of view, (since the predicate is true if any axiom is true), the Prolog execution mechanism always tries to prove predicates in the left-to-right order in which they appear in the body of a clause, and similarly it always selects axioms to be used to prove a predicate in the lexical order in which they appear. Thus, by the ordering of predicates in a clause body and of clauses within a procedure, the programmer can affect the order in which the Prolog system attempts to find proofs. In this way,

therefore, a programmer can influence the way in which a proof is attempted and thus determine, to an extent, the efficiency of a solution. For example, where a proof is recursive, the clauses representing the base-case axioms could be placed before the recursive axioms, so that non-recursive proofs are considered before recursive ones. For example, the definition of membership of a list represented by the clauses:[6]

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

has the base case as the first clause.

Unfortunately, the rather simple-minded way the clauses are chosen means that the programmer is *required* to exercise control over program execution—otherwise, proofs of interest might not be found even when they are implied by a logical interpretation of the clauses. Consider, for example, proving the query `member(A,B)`. By considering the first of the two clauses above, the Prolog interpreter will prove the query by unifying `X` with `A` and `[X|_]` with `B`, yielding an immediate proof. However, suppose the clauses were in the other order:

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

Now, by considering the first clause, the interpreter will prove `member(A,B)` on condition that it can prove `member(A,_)`, leading to infinite regress. Here, therefore, is a situation where the interpreter is unable to prove something that is a logical consequence of the axioms. It follows from this that a Prolog programmer must give consideration both to the logical description of the problem and also to the way in which the Prolog evaluation mechanism operates. This is captured neatly in Kowalski's celebrated aphorism: `algorithm`

[6]Here, the square brackets denote a list structure, and the underscore represents an *anonymous variable*—a variable that occurs just once in the clause and—since it is not referred to again—needs no name.

`= logic + control` [44]; the desired algorithmic behaviour is achieved by controlling the way in which deductions are performed on the logic.

Other aspects of a Prolog program not captured by treating it simply as a set of axioms include deficiencies due to the simplified form of unification used, (the *occur check* is omitted, as mentioned on page 9), the use of the 'cut' symbol, shortcomings in the treatment of negation, the reliance on predicates with side effects.[7]

For a thorough introduction to Prolog, the interested reader is referred to [22] or [11]. More advanced treatments can be found in [54] and [65].

It could be argued that Prolog is neither a proper logic nor a proper programming language; however, in many situations it does offer the programmer the opportunity to develop solutions to problems by considering logic and control somewhat independently of each other.

1.2 Example

In this section, an adaptive variant of mergesort called *runsort* is described that demonstrates the ease—one might even say the *elegance*—with which a fairly complex algorithm can be expressed and implemented. For a detailed analysis of runsort, including a comparison with a similar sorting algorithm called SAMsort [53] developed by Richard O'Keefe, please refer to [10].

In Prolog, a simple formulation of mergesort would consist of a predicate to split the incoming list of n items into a list of n singleton lists followed by predicates to merge these into $n/2$ lists of ordered pairs, in turn to be merged into $n/4$ lists each of four ordered items, and so on until just one ordered list remains. It is, however, a straightforward matter to make the algorithm adaptive by splitting the incoming list into lists of the *long runs* [42] present (see [32] for a survey of adaptive sorting algorithms). The code for this

[7] See [54] for an extensive treatment of these and related issues.

adaptive mergesort, termed 'runsort', is presented below.

Firstly, the following code defines `runsort/2` as a predicate calling the `split/2` predicate followed by the `merge_phase/2` predicate:

```
runsort([],[]).
runsort(X,Y) :-
        nonvar(X),
        split(X,Fragments),!,
        merge_phase(Fragments,Y).
```

The `split/2` predicate identifies the start of rising or falling runs in the input list and calls `splitRisingRun/4` to append the rest of a rising run `R` to the two items comprising the start of the the run, `X` and `Y`. If a falling run is detected, `splitFallingRun/5` is called to prepend the rest of the falling run, in reverse order, to the two items `Y` and `X`. Thus, the `split/2` predicate breaks the incoming list into lists of each long run, reversing the order of falling runs to return them in ascending order.

The `split/2` predicate is:

```
%split the incoming list into sublists,
% one for every long run
split([],[]).
split([X,Y|R],[[X,Y|L]|Z]) :-
        X@=<Y,
        splitRisingRun(R,S,L,Y),!,
        split(S,Z).
split([X,Y|R],[T|Z]) :-
        splitFallingRun(R,S,[Y,X],T,Y),!,
        split(S,Z).
split([X],[[X]]).

%identify the extent of a run that is known to be rising
% (i.e. non-descending)
splitRisingRun([X|R],S,[X|L],K) :-
        X@>=K,
        !,
        splitRisingRun(R,S,L,X).
```

```
splitRisingRun(R,R,[],_).

%identify the extent of a run that is known to be falling
% (i.e. descending) and return a list
% with the run reversed, i.e. in ascending order
splitFallingRun([X|R],S,Ti,To,K) :-
        X@<K,
        !,
        splitFallingRun(R,S,[X|Ti],To,X).
splitFallingRun(L,L,B,B,_).
```

The merging phase of the algorithm is implemented using the predicates `merge_phase/2` to repeatedly call `merge_pass/2` to `merge/3` all pairs of lists until just one list remains.

The `merge...` predicates are:

```
%merge sublists until just one sublist remains
merge_phase([],[]) :- !.
merge_phase([X],X) :- !.
merge_phase(X,Y) :-
        merge_pass(X,A),!,
        merge_phase(A,Y).

merge_pass([],[]).
merge_pass([X],[X]) :- !.
merge_pass([X,Y|R],[Z|A]) :-
        merge(X,Y,Z),!,
        merge_pass(R,A).

merge([],X,X) :- !.
merge(X,[],X) :- !.
merge([X|Y],[A|B],[X|R]) :- X@<A,!,merge(Y,[A|B],R).
merge(Y,[A|B],[A|R]) :- merge(Y,B,R).
```

1.3 The Origin of Prolog

As previously mentioned, Prolog was developed by Alain Colmerauer and Philippe Roussel at Université Aix-Marsellies. According to Jacques Co-

hen's account [23], Colmerauer had been interested in text understanding using logic deduction. While Alan Robinson's paper on resolution and unification [57] provided a theoretical basis for a language such as Prolog, it was extremely inefficient as a computational mechanism. Kowalski and Kuehner had developed a variant of linear resolution called SL resolution [46] and so Colmerauer invited Kowalski to visit Marsellies in the summer of 1971. As a result of that visit and another in the spring of 1972, the foundations of the language were laid, viz. the use of SL resolution on definite Horn clauses. Together, these features lead to the now-familiar Prolog notation, to goal-directed deduction by means of inference steps implemented using resolution, and to the use of backtracking to simulate non-determinism.

With the experience of Colmerauer and Roussel's first implementation behind them, the second implementation of Prolog by Gérard Battani and others [5] included practically all the features of 'modern' Prolog, including the cut and including operator definitions. This implementation was written in a combination of Fortran and Prolog. In designing Prolog, the Marseilles group appear to have chosen to make the language similar as possible to a high level language while still maintaining its claim as a logic programming language. Those choices were:

- To base Prolog on Horn Clause Logic. This allows resolution to be used as a means of inference;
- To omit the 'occur check'. Formally, resolution of two terms is forbidden if one term occurs in the other; hence 'full' resolution must include a check called the 'occur check' for this eventuality. If the occur check is omitted, resolution is considerably faster. Even though resolution without the occur check is unsound, Prolog omits the occur check;
- To use left-to-right subgoal satisfaction. Subgoals are satisfied in the order they appear in the clause body. If subgoal satisfaction is regarded

as a procedure call, then the body of a clause is similar to the code in a normal procedure, containing sequences of calls to other procedures;

- To use depth first satisfaction of goals. Where a goal reduces to subgoals, these subgoals are recursively satisfied before attempting to satisfy any other goals. This means that if goals are regarded as procedures, depth first satisfaction become depth first procedure calling, which is exactly what happens in a conventional language: if a procedure calls another, the subsidiary procedure must fully execute before its caller continues;

- To simulate non-determinism using backtracking. In the course of satisfying a goal, a number of clauses may be found to be suitable for evaluation. A non-deterministic machine will pick a clause that will lead to a proof. In Prolog, non-determinism is simulated by choosing the first suitable clause that occurs lexically in the program text, and by storing a reference (a 'choice point') to the alternatives. If it later transpires that the clause that was chosen does not lead to a proof, program execution is said to 'fail' and is unwound back to the point at which the clause was about to be chosen, but this time the next alternative is chosen, and normal operation resumes. This unwinding of program execution is called backtracking, and can occur repeatedly until all the choices stored in choice points have been tried. Choice points are stored in last-in-first-out order, and, on backtracking, the most recently made choice is revisited first. If all the choices in the most recent choice point are tried and found unsatisfactory, further backtracking will revisit the next older choice point, and so on. (It should be noted that the backtracking-on-failure mechanism does not fully simulate non-determinism. So long as the 'wrong' choice of clause eventually results in failure, causing backtracking to occur, then back-

Prolog Feature	*Imperative Language Feature*
set of clauses	program
predicate; set of clauses with same name and arity	procedure definition; nondeterministic case statement
clause; axiom	one branch of a nondeterministic case statement; if statement; series of procedure calls
goal invocation	procedure call
unification	parameter passing; assignment; dynamic memory allocation
backtracking	conditional branching; iteration; continuation passing
logical variable	pointer manipulation
recursion	iteration

Table 1.1: Some correspondences between features of Prolog and a conventional High Level Language (HLL). (From [71]).

tracking will simulate non-determinism. However, if the wrong choice does not lead to failure, backtracking will not occur, and the possibly correct alternative choice will not be made.)

Prolog has many points of correspondence with conventional computer languages—see Table 1.1 for a summary.

1.4 Early Implementations

Colmerauer and Roussel's original interpreter [58], written in Algol-W, was followed by Battani and Meloni's Fortran-based interpreter [5].

A Prolog interpreter was developed in Hungary in 1975 by Péter Sz-

eredi, based on an example showing how Prolog might be implemented in a talk given by David H. D. Warren [66]. Maurice Bruynooghe's Pascal-based Prolog interpreter [12] appeared in 1976. Grant Roberts also developed an interpreter at the University of Waterloo, Canada, in 1977 [56].

The first Prolog compiler was developed by David H. D. Warren, producing DEC10 Assembly Language [72, 73].

The very well-known C-Prolog was an interpreter developed at Edinburgh in 1982 by Fernando Pereira, Luís Damas and Lawrence Byrd [55]. It is based on EMAS Prolog, a system completed in 1980 by Luís Damas. According to [71], a number of Prolog implementations appeared from Edinburgh over the next few years, in particular Prolog-X and NIP (New Implementation of Prolog). DEC10-Prolog was extensively modified sometime around 1980, incorporating Last Call Optimisation [74, 76].

In 1983, David Warren published his account of what is now known as the Warren Abstract Machine [75]. The WAM, as it became known, is clearly an extension of his DEC-10 work. Structure copying is employed for representing terms and registers are used to hold arguments and terms. The WAM has formed the basis for many, if not most, subsequent Prolog implementations.

A comprehensive account of developments in Prolog implementations is given in Peter Van Roy's report *1983–1993: The Wonder Years of Sequential Prolog Implementation* [71]. In that work, Van Roy traces the themes in Prolog implementation in these years: improvements in compilation techniques for unification, for clause selection and backtracking; compilation to native code; optimisations arising from global analysis and abstract interpretation; development of alternative execution models, including the Vienna Abstract Machine [47] and BinProlog [67].

Virtually all of these start with the WAM, or are based on structure copying, but interestingly enough, David Warren himself, in a foreword to Hassan Aït-Kaci's tutorial on the WAM [1], says:

Although the WAM is a distillation of a long line of experience in Prolog implementation, it is by no means the only possible point to consider in the design space. For example, whereas the WAM adopts "structure copying" to represent Prolog terms, the "structure sharing" representation used in the Marseilles and DEC-10 implementations still has much to recommend it'

1.5 Implementation Issues

From the start, speed and memory consumption were the main concerns of Prolog implementers, e.g. see [16] for coverage of some of these issues. The move from interpreter-based to compiler-based implementations was intended to address both issues.

Two ways of representing dynamically constructed data structures evolved: *structure sharing* and *structure copying*. Memory was conserved by treating short-lived and long-lived memory allocations separately, and by recovering certain categories of deallocated memory automatically. Garbage collection was introduced to recover unused memory. Speed was enhanced by specialising unification routines as much as possible so as to minimise the amount of processing required. Some of these issues are considered in the following sections.

1.5.1 Dynamic Data Structures: Structure Sharing and Structure Copying

A peculiar feature of Prolog as a computer language is that it has no explicit data constructs. In fact, no distinction is drawn between 'program' and 'data' in Prolog. Instead, clauses fill the role of both program and data. Structured terms, also called compound terms, fulfill the role of dynamic data structures

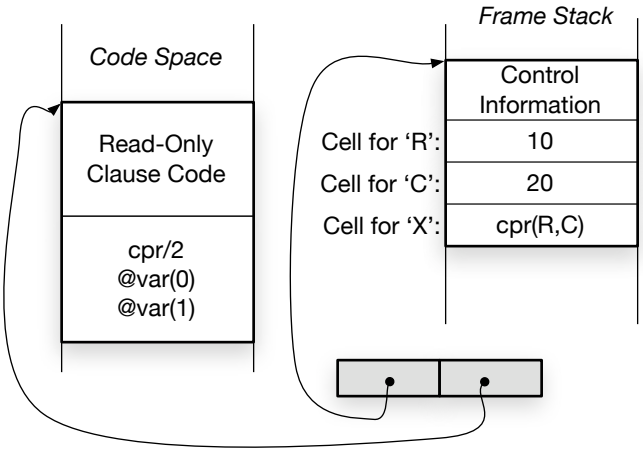

Figure 1.2: A clause instance, represented by a pair of pointers, shown in grey—one to the fixed code of the clause, the second to a frame of data where, along with control information, the values assigned to variables of that particular instance of the clause are stored.

in a conventional language. For example, the clause `makeComplex(R,C,X):- X = cpr(R,C)` contains one compound term, `cpr(R,C)`. A call to this clause, passing a variable in for `X` results in the return of a newly-constructed data structure, the compound term `cpr(R,C)`. The representation technique used for compound terms—especially compound terms like this one, that are constructed by a running program—is an important part of any implementation.

When the goal `?- makeComplex(10,20,T)` is executed, (assuming the clause above is part of the program), an instance of the clause is created. Conceptually, the clause instance comprises two items: the fixed code for the clause and a frame of memory locations for the values of the variables and the control information such as return address, etc. The clause instance could be represented by a pair of pointers (see Figure 1.2), one to the frame of memory locations and the second to the clause code.

Once the goal has unified with the head of the clause instance, `R` has the value `10` and `C` has the value `20`, both values being stored in the appropriate

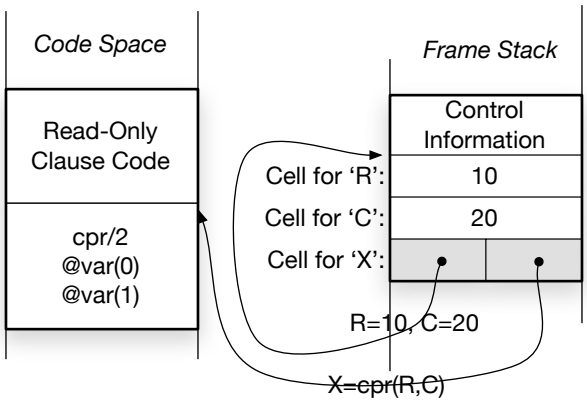

Figure 1.3: A new structure, constructed by structure sharing, is represented by two pointers shown in grey—one to the structure's 'skeleton' which is part of the clause code, the second to a frame of data where the values assigned to variables of that particular instance of the structure are stored.

part of the instance's frame. The variable `T` has been bound to the variable `X`. At this point, the call `X = cpr(R,C)` is executed, and a new compound term, `cpr(10,20)` must be constructed and unified with `T`.

If the implementation uses *structure sharing*, the term will be represented by a pair of pointers (please refer to Figure 1.3) , often called a *molecule*: the first pointer will point to the frame of memory containing the cells that hold the values of the instance variables, and the second pointer will point to that part of the clause code that represents the structure `cpr(R,C)`. In fact, the structure instance is almost identical in form to the clause instance to which it is related. The difference is that the clause instance's code pointer points to an executable part of the clause code, whereas the structure instance's code pointer points to that part of the clause code that represents the structure of the term—the so-called *skeleton* of the term.

If, on the other hand, the implementation uses *structure copying*, then the structure will be represented (please refer to Figure 1.4) by constructing a completely new copy of the term with the new values of variables in place.

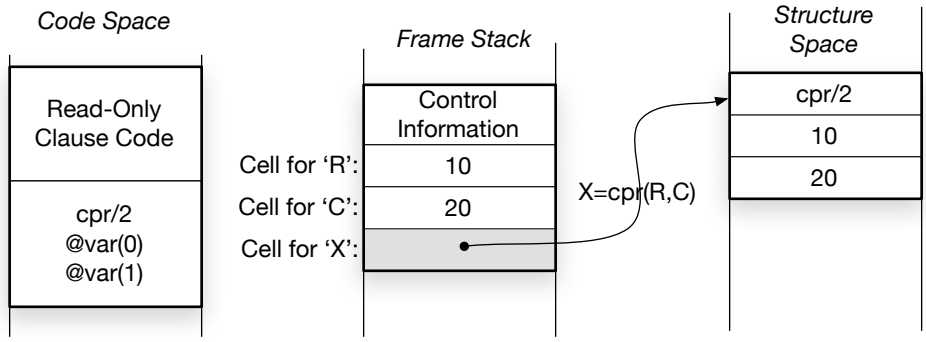

Figure 1.4: A new structure is constructed in a separate area, here called the Structure Space. The values of any variables that are known are also copied into the structure. The structure is referenced with a simple pointer.

In other words, the Prolog interpreter translates the structure instance into a new copy of the term instance, where instantiated variables are replaced by the values to which they are bound. This requires an area in which the new copy of the term can be constructed. The new structure instance may be referred to with a single pointer.

Each representation technique has its own advantages and disadvantages, to be considered in later sections of this work. Using structure sharing, for example, unification of a constructed term with a variable is fast, but if a structure instance is needed after the clause instance that it is part of has terminated, the part of the clause instance's frame containing its variable value cells must be kept in existence. This complicates the memory management of structure sharing implementations. Additionally, a variable could potentially have to be large enough to accommodate the two pointers of a molecule.[8] If structure copying is used, clause instance frames can be deleted on termination of the clause instance, but unification of a constructed term with a variable may be slow due to the copying needed.

[8]In MProlog, this problem is avoided by constructing molecules as needed on the global stack and pointing to them from variables [43].

Of the two techniques, structure sharing is the older in Prolog. According to Warren [72], the structure sharing technique [9] was first used in Prolog by Battani and Meloni [5] in 1973. Warren's 1977 implementation [72] of DEC-10 Prolog [7] used structure sharing, as did Roberts' [56] implementation in the same year.

Structure copying was proposed separately at the Workshop on Logic Programming in Debrecen, Hungary, in 1980 by Bruynooghe [13] and Mellish [51]. In 1983, Warren published his description of the WAM, the Warren Abstract Machine [75], which uses structure copying. Virtually all later Prolog implementations use structure copying, though Open Prolog uses structure sharing.

1.5.2 Memory Management

One of the key developments in Prolog implementations was the incorporation of garbage collection. The earliest implementations, lacking garbage collection, relied on various programmer stratagems to regain memory, for example by backtracking from time to time, having stored partial results from which to resume operations. Last Call Optimisation (LCO) and garbage collection were studied intensively [13, 14]. DEC-10 Prolog was the first to introduce garbage collection, using a mark-compact algorithm[9] on the global stack [72], and LCO was introduced in the later version of DEC10-Prolog [74, 76].

1.5.3 Early Developments

One of the visitors to Marseilles in early 1974 was David H.D. Warren. When Prolog 'escaped' from Marseilles, in the form of copies of Battani *et al's* 1973 implementation, one of its destinations was the University of Edinburgh,

[9]For a survey of garbage collection topics and techniques, see [41]

where it was installed on a PDP10 by David H.D. Warren[10] with the assistance of the original implementers [25, p. 9]. It would be difficult to overstate the importance of David H.D. Warren's own implementation work, starting with a description of the architecture and implementation of a high performance Prolog implementation for the DEC 10 family of machines [72, 73]. DEC-10 Prolog was the first implementation to actually compile Prolog into machine code, and was very fast and memory-efficient.

There were two major innovations in DEC-10 Prolog. The first was to use each argument in a clause's head to generate instructions that would perform specialised unification with the corresponding goal argument. For example, if a head argument was an integer, then code would be generated to unify an integer of that specific value with the corresponding goal argument, thereby avoiding the overhead of a general unification routine. The second major innovation was to categorise all the variables in a clause on the basis of their usage and possible lifetimes. Variables that could be shown to have short or very short lifetimes could be allocated space in a 'local stack'—roughly equivalent to the stack in a conventional language—where the space occupied by them could be easily recovered without expensive garbage collection. All other variables were allocated space in a 'global stack'—roughly the same as a heap in a conventional language. In some cases, analysis could determine that a variable was merely a placeholder; in that case, no code was generated for it at all.

An [unnamed] implementation of Prolog was developed around the same time in Hungary by Péter Szereidi. This was superseded by MProlog, which was based on the PLM—see [66, 33].[11]

[10]There are two David Warrens in the logic programming community; the other one is David S. Warren of SUNY.

[11]The MProlog compiler is based on Warren's later work [74, 76] where argument registers and Tail Recursion Optimisation were introduced. MProlog has been enhanced and refined over the years [43].

Up to this time, the technique of structure sharing was used to represent constructed structures. Bruynooghe and Mellish published work on the use of structure copying in 1980 [13, 51] and Warren published the design of the *Warren Abstract Machine (WAM)* in 1983 [75]. The WAM used many of the techniques developed in his 1980 modification of DEC-10 Prolog, such as argument registers, but, perhaps more significantly, it used structure copying to represent constructed structures. The history of Prolog implementations from this point on is very well treated in [71], to which the interested reader is referred.

Chapter 2

The DEC10-Prolog Machine

2.1 Introduction

As mentioned in Section 1.4 (page 17), the development of DEC-10 Prolog was one of the most important events in the history of Prolog, and was very influential on later Prolog implementations, including Open Prolog.

DEC-10 Prolog is a compiler-based Prolog implementation for the DEC-10 [7], and the basis for the implementation is the work described by David H.D. Warren in his thesis [72, 73]. Source code is compiled into the machine code of a Prolog-oriented machine architecture called the PLM. From this, the Prolog compiler generates DEC-10 macro assembler code. Structure sharing is used to represent constructed compound terms. Clauses are compiled individually, and the clauses that comprise a procedure are linked using `try` instructions, preceded by an `enter` instruction.[1] Figure 2.1 shows this arrangement schematically for a procedure comprising three clauses.

The arguments of a clause's head are encoded into executable code—the *head code*. Each argument, called a *level 0 argument*, is compiled into host

[1]For simplicity, this discussion relates to unindexed clauses. The indexing scheme devised for the PLM is discussed later.

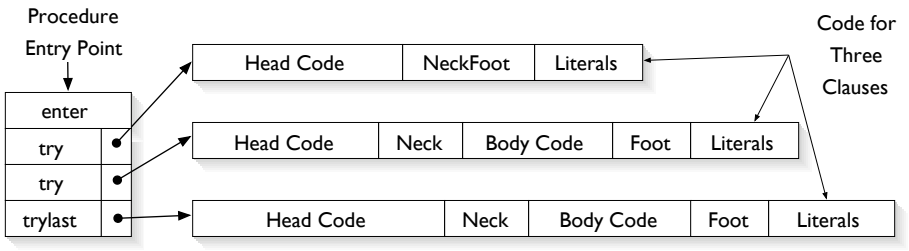

Figure 2.1: Schematic of a PLM procedure. The sequence of `try` instructions links the machine code for each separate clause together. The `enter` instruction partly initialises the clause instance's environment.

instructions that will attempt to unify it with a corresponding argument in the current goal. If an argument is a structure, then the arguments of the structure—*level 1 arguments*—are similarly compiled into machine code.

Execution of the head code unifies each head argument with its corresponding goal argument and creates part of a new instance of the clause as a side effect. This avoids separately instantiating a clause instance before attempting to unify the current goal with it. The *neck code* following the head code completes the construction of the clause instance's environment once the head code has successfully completed. Goals in the body of the clause are compiled into procedure calls, and cut symbols are compiled using a special `cut` instruction. Thus, the *body code* is a sequence of `call` and `cut` instructions. The body code is terminated by the `foot` instruction which is the equivalent of the restore-and-return instructions at the end of a conventional procedure.

Certain common sequences of instructions are replaced by specialised combinations. For example, a unit clause has no goals in its body, so the neck instruction that would complete the clause instance's environment would immediately be followed by a foot instruction which might discard it. According to Warren, combining the two instructions in a `neckFoot` instruction saves

Name	*Lifetime Ends*	*Criterion*
Global	Backtracking.	Occurs in a skeleton.
Local	Procedure completed successfully and determinately, i.e. no choices remain within the procedure.	Multiple occurrences with at least one in the body and none in a skeleton.
Temporary	Completion of unification with the head of the clause.	Multiple occurrences, all in the head of the clause and none in a skeleton.
Void	None.	A single occurrence, not in a skeleton.

Table 2.1: Classification of Variables in the PLM.

space and execution time.[2]

Along with head, neck, body and foot code, each clause also gives rise to `try` instructions. The `try` instructions of all clauses whose heads have the same principal functor are linked for fast access to individual clauses within the procedure. The last clause in the procedure has a `tryLast` instruction rather than a simple `try`.

Structures and goal arguments are compiled into non-executable literals and stored as data after the `foot` instruction at the end of the clause.

2.2 Variable Lifetime Analysis

Considerable attention is given in the PLM to reducing the memory requirements of running programs. One of the main innovations of the PLM is the analysis of variables into one of four categories depending on their likely lifetime. The four categories are listed in Table 2.1, (from [72, p53]).

Broadly speaking, there are two categories of variables, *global* and *local*.

[2]This is not borne out in Open Prolog; the equivalent instruction has a very small effect on program behaviour.

- Variables that appear as part of a structure are automatically classified as global.

- Variables that appear only once, and not in any structure, are *voids*. Voids are placeholders, because their values are never referenced or used, so no memory is allocated for them.

- Variables that occur only in the head of a clause, and not in any structure, are classified as *temporary*. Once unification of the calling goal with the head is complete—i.e. by the time the `neck` instruction is executed—temporary variables are no longer needed, since they don't occur in the body of the clause. Storage allocated to temporaries can therefore be deallocated as soon as head unification is complete.

- Finally, variables that occur in the head or body of the clause, but don't occur in any structures, are local variables.

To take advantage of the lifetime analysis of variables, the PLM has a *global stack* for global variables and a *local stack* for local variables. These and other data areas are examined later in this thesis.

2.3 Variable First Usage Analysis

Variables occurring in the head of a clause are also analysed by usage. As previously mentioned, the head instructions of a clause unify with the current goal while simultaneously constructing a new clause instance. The first occurrence of a variable in the head is a special case, since it is known that the variable has never been used before. All that is required is to bind it to its matching term in the goal—it doesn't need to be initialised first. If it happens to be classified as a local, then this operation reduces to simple assignment—the corresponding goal argument is simply copied into it. This

is a significant simplification of the general case of unification, and Warren attributes part of the speed of DEC-10 Prolog to it.

2.4 Data Representation

The PLM uses structure sharing to represent constructed compound terms. These are represented by a pair of pointers—a *molecule*—one pointing to a skeleton literal and the other to a frame of memory locations containing the values held by the variables referenced from the literal. The representation of constructed terms is thus different to the representation of literals or source terms. The PLM therefore has one set of representations for literal terms and another set for constructed terms, i.e. terms generated by running programs and assigned to variable cells.

2.4.1 Representation of Literals

There are essentially four kinds of literals to be represented in DEC-10 Prolog: *variables*, *numbers*, *atoms* and compound terms or *skeletons*. Variables are classified into one of five categories: `var`, `global`, `local`, `temporary` and `void`. A `var` is the identification used for a global variable wherever it occurs within a structure. Temporary variables are treated as locals, and the PLM relies on a combination of DEC-10 addressing modes and tags to distinguish between the resulting seven types.

Each of the seven types is represented initially by a DEC-10 `XWD`, which is a 36-bit word that can contain an effective address. The layout of an `XWD` is shown in Figure 2.2. As may be seen, the effective address can include an index register specification. If the upper halfword is zero, the lower halfword is treated as an absolute address. Particularly worth noting is the `I` bit. If this bit is set, the `XWD` is taken as a memory-indirect reference to another

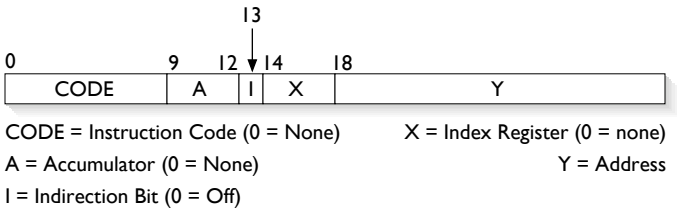

Figure 2.2: Layout of a DEC-10 instruction.

`XWD`, which might also have its `I` bit set, indicating that it, in turn, is another memory-indirect reference. Thus, the DEC-10's `XWD` offers direct support for reference chains.

As we will see below, three PLM registers, `Y`, `X` and `X1` are index registers for `var`, `local` and `global` variables respectively.[3] Thus, the layout for all seven distinguished literals is as shown in Figure 2.3.

An atom is represented by a reference to a tag-and-data word, where the data, according to Warren [73, p6] 'identifies the atom' and distinguishes it from any other. Similarly, the data part of an integer is its binary value, in 2's complement form if negative. A structured term or 'skeleton' is represented by a reference to a sequence of terms representing the principal functor, `skel(I)`, followed, recursively, by representations of the subterms, in order. Nested structures are thus accommodated *via* out-of-line references.

The representation is notable for two reasons. First, inline operands are all the same length. This simplifies many operations, such as finding the n-th argument. Second, the representation scheme fits very snugly into the DEC-10 architecture; by making such efficient use of the processor's unusual memory-indirect effective address calculation mechanism, access to

[3]The fact that some PLM registers have the same names as fields in an `XWD` is accidental—there is no connection.

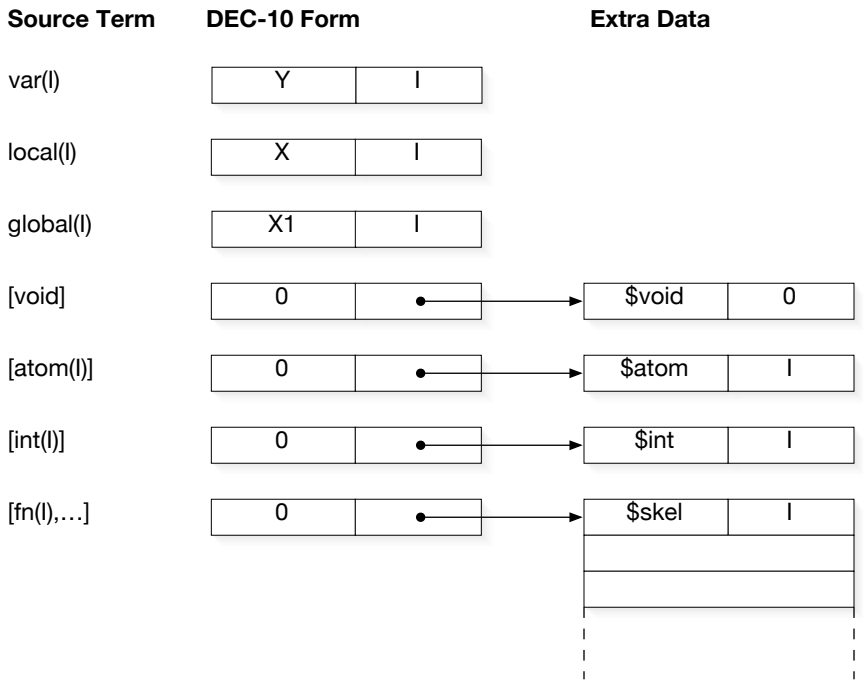

Figure 2.3: Literals in DEC-10 Prolog. All representations are effective addresses in the DEC-10 architecture, and are evaluated prior to use. A structure is represented by a pointer to the `$skel` followed by the principal functors index followed by the subterms of the structure.

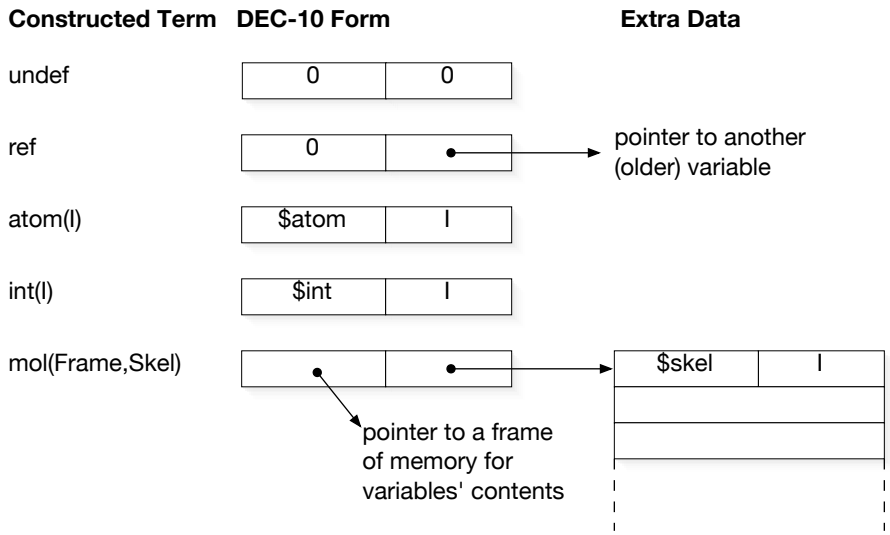

Figure 2.4: Constructed Terms in DEC-10 Prolog. A structure is represented by a 'molecule' of two pointers. The first pointer points to a vector of memory called a frame in which the values of variables are stored. The second pointer points to the 'skeleton' or fixed structural part of the term.

arguments is fast and efficient.[4] Every literal is uniformly defererenced to access its actual value, considerably simplifying and speeding the resulting code.

2.4.2 Representation of Constructed Terms

Constructed terms are values given to variables during program execution. In the PLM, they can have one of five types of value: undefined, bound to another variable, unified with an atom, unified with an integer and unified with a constructed compound term or molecule. Figure 2.4 shows the layout of constructed terms.

[4]Warren states that the possibilities afforded by the DEC-10 architecture inspired the development of DEC-10 Prolog.

2.5 Data Areas

The PLM has four data areas: the *code area*, the *global stack*, the *local stack* and the *trail*.[5] The code area contains the program code and literals, and is 'read only' once the program has been loaded. The trail is a push-down-list used for resetting variables on backtracking, and its use is described later in this report.

The global and local stacks are managed jointly to provide memory space for representing clause instances, including housekeeping information and space for variables. These areas are called stacks because they are allocated and deallocated on a last-in-first-out basis as clause instances are created and destroyed (on backtracking).

A clause instance's global variables, likely to have long lifetimes because they appear in structures, are allocated space in a *global frame*, and space for the rest of the instance's environment—local and temporary variables and environmental information—is allocated in a *local frame*.

Local frames and the local stack are managed much the same as the stack in a conventional language implementation. Local frames are allocated space at the top of the stack, on a conventional last-in-first-out basis. When a clause instance is finished with or backtracked over, its local stack frame is deallocated.

Global frames are managed differently. A global frame is allocated on top of the global stack when a clause is instantiated and is deallocated on backtracking, but it is not deallocated when the clause instance finishes. It cannot be deallocated because it contains all the variables that appear in structures in the clause instance that are therefore liable to appear in term instances or 'molecules'. These molecules may be returned to the clause instance's caller and some may be referenced during subsequent execution of the program.

[5] A fifth area for storing functor and atom definitions, is necessary for a functioning Prolog system, but is not described in the report.

Thus, the memory space they occupy in the global frame must remain allocated after execution of the clause instance has terminated. Unreferenced global frame space can be recovered by garbage collection, detailed later.

Overall, therefore, splitting a clause instance's environment between a global and a local frame allows much of the instance's memory allocation, specifically the local frame, to be recovered as soon as evaluation of the goal had been completed deteministically. Only the necessary parts of a clause instance's environment need to be preserved after the clause instance terminates, and they are the global variables. This innovation greatly reduces the PLM's appetite for memory.

2.6 Registers

Warren details eleven PLM registers. Of these, three pairs point to the tops of the stacks, to the frames of the current goal and to the frames of the most recent choice point. That is, `V` and `V1` point to the tops of the local and global stacks, `X` and `X1` point to the local and global frames of the current goal, and `VV` and `VV1` point to the local and global frames of the latest choice point respectively. Of the remaining five PLM registers, `TR` points to the top of the Trail, `PC` points to the current instruction, `A` points to the arguments of the current goal and the continuation, and the pair of registers `B` and `Y` are used to represent a molecule undergoing unification.

In addition, Warren classifies five more registers as part of the DEC-10 implementation.[6] The `FL` register comes in to play if there are alternative clauses to consider to the clause chosen to satisfy the current goal.

[6]It appears that these registers are used to improve performance on the DEC-10 but are not a vital part of the design of the PLM.

PLM Registers	
Register(s)	Function
V & V1	*Current Clause Frames.* Pointers to tops of local and global stacks respectively. Clause instance frames are constructed at the top of the stacks.
X & X1	*Current Goal Frames.* Pointers to the local and global frames of the Current Goal.
VV & VV1	*Latest Choice Point Frames.* Pointers to the local and global frames of the most recent choice point.
TR	*Top of Trail.* TR is a pointer to top of the trail data area.
PC	Current Instruction
A	Pointer to the arguments of the Current Goal and Continuation.
B & Y	Pointers to the skeleton and environment of a molecule involved in unification.
Extra Registers for the DEC-10 Implementation	
FL	Alternative to Current Clause (if there is one)
T & B1	Argument pointers to unification routine
C	Return address for a runtime routine
R1 & R2	Temporary result registers

Table 2.2: PLM Machine Registers.

2.7 Instructions

The Instruction Set of the PLM can be divided into a number of sets, listed in Table 2.3 and summarised in Appendix A.

The PLM is a compiler target; that is, the compiler generates PLM code from Prolog source, and the PLM code is macro-expanded into DEC-10 machine code.

Clause Selection Instructions	
`enter`	
`try(L)`	`tryLast(L)`
Unification Instructions	
`uvar(N,F,I)`	`uref(N,F,I)`
`uatom(N,A)`	`uint(N,I)`
`uskel(N,S)`	
`uvar1(N,F,I)`	`uref1(N,F,I)`
`uatom1(N,A)`	`uint1(N,I)`
`uskel1(N,S)`	
Other Head & Neck Instructions	
`neck(I,J)`	`init(I,J)`
`localinit(I,J)`	`ifdone(L)`
Body Instructions	
`call(L)`	`cut(I)`
Foot Instructions	
`foot(N)`	`neckfoot(J,N)`
`neckcut(I,J)`	`neckcutfoot(J,N)`
`fail`	

Table 2.3: PLM Instructions

2.8 Operation

The PLM machine operates quite like a conventional machine. In a conventional language implementation, the environment of a procedure instance that is about to make a procedure call would be at the top of the frame stack and the environment of the new procedure instance would be built on top of it. When the new procedure terminates, its environment would be discarded and the caller's environment would once again be on top of the stack. Something like this also happens with the Prolog implementation on the PLM—when a new procedure is called, its environment is constructed on the top of the stacks. A crucial difference arises when the new procedure exits: its stack frame is *not* discarded if alternative solutions are available for that procedure call. In fact, in the PLM, an environment is discarded only after there is no possibility of executing the goal again—i.e. if execution of the goal is determinate or if backtracking removes the goal itself.

The possibility that a goal's local frame will be left on the stack after execution of the clause instance has exited gives rise to a further peculiarity: the frame of the caller of a goal may not be at the top of the local stack when the call is made—it may be buried under local stack frames belonging to goals already executed.

2.8.1 Goal Execution

When a goal is called for execution, it becomes the *current goal*. As a subgoal of a clause instance, the current goal's local and global frames are those of the current goal's clause instance. Figure 2.5 depicts the data areas at this time. `V` points to the top of the local stack, `V1` points to the top of the global stack and `TR` points to the top of the Trail. The environment of the current goal is referenced by `X` and `X1`, and it may or may not be the topmost environment in the stacks. The environment of the most recent choice point (i.e. the goal

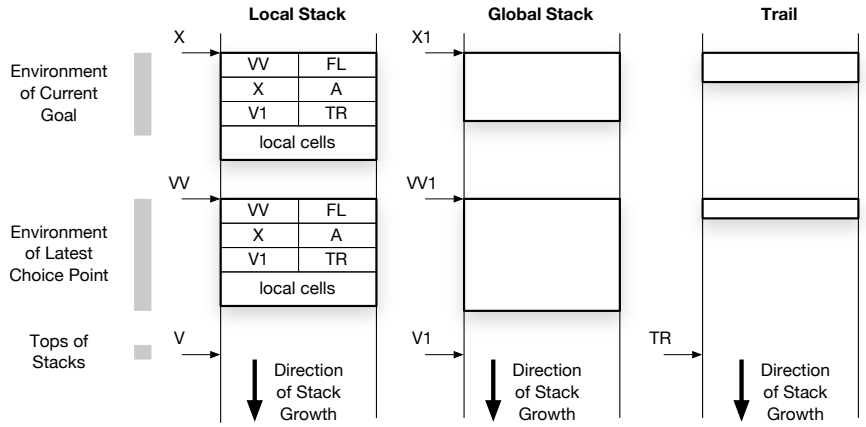

Figure 2.5: PLM data areas just as a goal is called for execution.

to which backtracking will go to if unification fails) is referenced by `VV` and `VV1`. The choice point could be later in the stacks than the current goal, as shown, or it could be earlier than, or coincident with, the current goal.

The machine is about to attempt to construct an instance of a clause whose head terms unify with the goal being called for execution. The environment for the new clause instance will be constructed on the top of the stacks. Thus, `V` and `V1`, as well as pointing to the tops of the local and global stacks, are also pointers to the frames of the new clause instance being created.

The `call` instruction firstly executes the `enter` instruction. The effect of this instruction is as follows:

- Copies of some PLM registers—`VV`, `X`, `A`, `V1` and `TR`— are stored in the new environment[7]

[7]Warren's philosophy was to delay 'housekeeping' functions as long as possible, in case a failure occurred that made the housekeeping unnecessary. So the `enter` instruction does the minimum housekeeping possible and initialisation is completed as execution of the head instructions progresses.

- The `VV` register is made equal to `V` and `VV1` is made equal to `V1`. `VV` and `VV1` point to the most recent choice point, so the significance of this action is to make this procedure call the most recent choice point.

Having partly initialised the new local frame, the `try` instruction for the first clause is executed. This sets the `FL` register to point to the next `try` instruction and then transfers program execution to the first instruction in the head of the first clause. If the clause is the last clause in the procedure, its `try` instruction is replaced by a `tryLast` instruction. The `tryLast` instruction sets the registers in the PLM to reflect the fact that there are no further choices for the current goal: the previous values of `VV` and `VV1`, (that specify the previous choice point), are retrieved from the environment where they were stored by the `enter` instruction. The effect of this is that if the last clause fails, backtracking will fall back to the previous backtrack point.

At this point, a clause instance has been partly created—a clause has been selected and an environment has been partly initialised. The goal arguments are pointed to by register `A` and the goal's local and global frames by registers `X` and `X1`; the instructions in the head code can then access goal arguments via these registers.

The PLM reserves two registers for dealing with arguments that are structures, registers `B` and `Y`, that correspond roughly to registers `A` and `X1`. Registers `B` and `Y` are used by *level 1 instructions* to access structure arguments and variables, respectively, and are initialised by the `uskel` instruction.

The `uskel` instruction is executed where a head argument is a structure. If the corresponding goal argument is a variable, and is unbound, it is assigned a constructed structure, i.e. a molecule, consisting of a pointer to the structure's literal and the new clause instance's global frame, register `V1`. If the goal argument is a bound variable, it must be bound to a molecule having the same functor as the head argument. Preparations are made for the execution of level 1 instructions by assigning register `B` to point to the

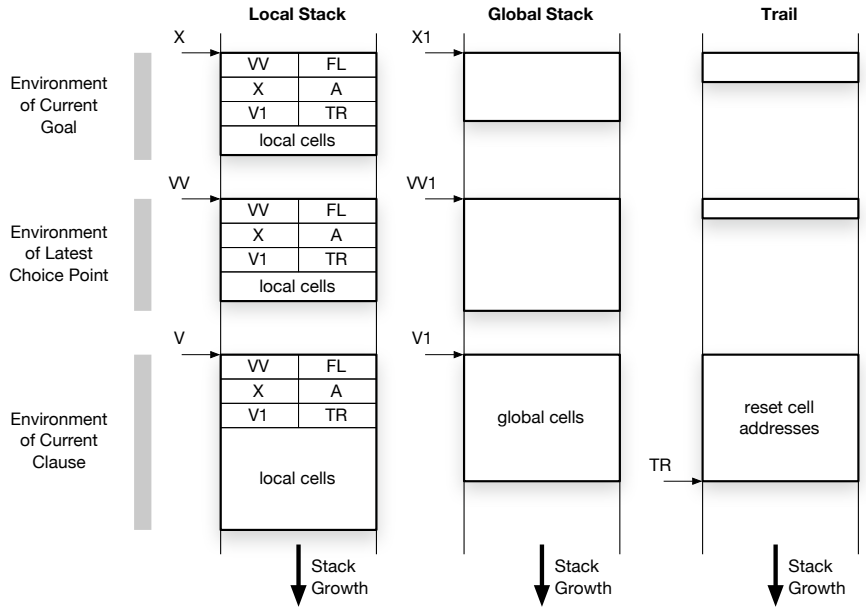

Figure 2.6: PLM data areas after the `enter` instruction.

molecule's arguments and register `Y` to its frame.

If the goal argument is a structure, it must have the same principal functor as the head argument, and, to prepare for the execution of the level 1 arguments, register `B` is assigned to point to the goal structure's arguments and register `Y` is made equal to the goal's global frame pointer `X1`.

The instructions in the clause head attempt to unify the clause instance with the goal, initialising newly-encountered variables as necessary. Structures containing uninitialised variables are preceded by `init` instructions whose function is to initialise just those variables. In this way, as the head instructions execute, more and more of the clause instance's environment is initialised. If all the head instructions execute successfully, (meaning that the goal unifies with this clause instance), all variables that occur in the clause head will be initialised. The `neck` and `init` instructions complete ini-

tialisation of the clause instance's environment by initialising any variables that occur in the body of the clause but not in its head and by copying in the remainder of the control information:

- The current goal pointers, `X` and `X1` are copied into the new environment as the new clause instance's continuation frame pointers.
- The current value of the `PC`.

Finally, the new clause instance is complete, and it now becomes the new 'current goal'. This is accomplished by copying `V` to `X` and `V1` to `X1`, by advancing `V` beyond the end of the new local frame and `V1` beyond the end of the global frame. The PLM is now ready to execute the first instruction in the body of the new clause instance.

2.8.2 Failure

If any of the head instructions fail to unify with their corresponding goal arguments, (meaning that this clause can not be resolved with the current goal), the PLM will backtrack to the most recent choice point—the goal whose environment is pointed to by `VV` and `VV1`. When backtracking occurs, the state of the machine is reset to the state it was at the time when the goal at the backtrack point was about to be executed. All frames constructed subsequent to the backtrack point are popped from the stacks, so that all the control information and variable values stored in these frames are discarded. However, those variables that were not in the discarded frames but that were instantiated by the execution of discarded clauses must be reset. Such variables are 'trailed' when they are instantiated, that is, a reference is put into the Trail (referenced by the `TR` register). Trailing is an expensive operation, so it is only done when necessary. When backtracking occurs, variables that must be reset can be identified in the Trail.

The PLM distinguishes two kinds of failure—*shallow failure* and *deep failure*. If the most recent choice point is the current clause instance (i.e. `VV` = `V`), then it is said to be a 'shallow fail'. In that case, restoring the environment is relatively simple:

- Variables that were trailed since the clause instance started execution are reset. These are easily identified, as their trail entries are above the value that `TR` had when it was saved by the `enter` instruction at the start of execution of the current goal. Thus, variables referenced in the trail between the current and the stored values for `TR` are reset and `TR` is then reset.
- Execution is transferred directly to the clause whose `try` instruction is referenced by the `FL` register.
- The Goal Argument Pointer `A` is reset from the environment.

No other registers need to be reset.

If the most recent choice point is not the current goal, then 'deep' backtracking occurs, and it is a more expensive operation than shallow backtracking:

- All stack frames later than the choice point's frames are discarded.
- All variables trailed since the choice point are reset.
- PLM registers are restored from the control information stored in the choice point's local frame.

2.9 Example

In this section, adapted from page 56 of [72], we illustrate the coding for the `member/2` procedure:

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

In the first clause, `X` and `L` are both global variables because they appear in structures. The code for the first clause is:

	Code	Source	Classification
clause1:	uvar(0,global,0)	member(X,	Code
	uskel(1,label2)	[	Code
	init(1,2)		Code
	ifdone(label1)		Code
	uref1(0,global,0)	X	Code
label1:	neckfoot(2,2)	)	Code
label2:	fn(./2)	[	Literal
	var(0)	X|	Literal
	var(1)	L]	Literal

The second clause has one local variable, `X`, and two global variables, `Y` and `L`. The code for the clause is:

	Code	Source	Classification
clause2:	uvar(0,local,0)	member(X,	Code
	uskel(1,label4)	[	Code
	init(0,2)		Code
	ifdone(label3)		Code
	uvar(1,global,1)	L]	Code
label3:	neck(1,2))	:-	Code
	call(member)	member(	Code
	local(0)	X,	Literal
	global(1)	L)	Literal
	foot(2)		Code
label4:	fn(./2)		Literal
	var(0)		Literal
	var(1)		Literal

2.10 Clause Indexing

Clause Indexing is another important innovation in the PLM that potentially speeds up clause selection and reduces memory requirements. Recall that

when a goal is called for execution, the clauses in the procedure are tried sequentially in the lexical order they appear in the program. In the PLM, the clauses of a procedure are indexed by their first arguments. The first argument of the goal is used as an index selector, so that only those clauses in the procedure whose first argument matches the goal's first argument are considered. In many cases of interest, just one clause will match, and that clause may be chosen at once without having to build a choice point.

While the idea of clause indexing is simple, its application is complicated because it must honour the sequential clause selection mechanism of Prolog. If the first argument of the goal is non-variable, and if every clause in the procedure has a distinct non-variable first argument, then the indexing mechanism will select the only possible clause, if it exists, or else it will fail. However, if the first argument of the goal is a variable, it could match the first argument of every clause, so each clause has to be considered in lexical order. Also, any clause whose first argument is a variable would have to be considered for any case.

The approach used in the PLM is to divide the clauses in a procedure into sequential *sections*. A lexical sequence of clauses where the first argument is a variable forms a *general section*, and a lexical sequence of clauses where the first argument is a non-variable forms a *special section*. A procedure can have any number of special and general sections alternating with one another. Sections are linked sequentially, and when a goal is called for execution, it traverses the sequence of sections.

The code for a general section begins by dereferencing the goal's first argument; the clauses are tried sequentially, and if none match with the goal, the next special section is entered, if it exists—otherwise a failure occurs.

The code for a special section consists of two parts, the *reference code*—used when the first goal argument turns out to be a reference to a variable—and the *non-reference code*, used when the first goal argument is a non-

```
call(X or Y) :- call(X).
call(X or Y) :- call(Y).
call(trace) :- trace.
call(notrace) :- notrace.
call(read(X)) :- read(X).
call(write(X)) :- write(X).
call(nl) :- nl.
call(X) :- ext(X).
call(call(X)) :- call(X).
call(true).
call(repeat).
call(repeat) :- call(repeat).
```

Figure 2.7: This is the procedure whose indexing code is illustrated in Figure 2.8.

variable.

If the section is a special section and the goal's first argument is non-variable, the non-reference code is used: a hash index may be computed from the goal argument and used to select those clauses in the section whose first argument could match it. Frequently, there is only one such clause, and that is selected immediately. If there is more than one such clause, each one is tried sequentially. If the goal's first argument is a variable, the reference code in the special section is used to try the clauses sequentially. If none of the clauses in a special section match the goal, then the following general section is entered, if there is one—otherwise a failure occurs.

A sample of code from [72] and listed in Figure 2.7 is translated into the (rather complex) indexing code shown in Figure 2.8.

Although this is a complex example, it has the merit of showing most of the features of the PLM indexing scheme. The indexing code comprises three sections: a special section for the first seven clauses, a general section for the eighth clause and a special section for the remaining four clauses.

The non-reference part of the first special section, being more than five clauses, uses a hash table whose size—8—is the next-highest power of two.

```
SSECT1          ssect(ref1,next) ;start of special section; ref1 points to the reference code

                switch(8)               ; 8-way hash table for 6 clauses
                case(label1)            ; note that only 5 entries are used
                case(label2)            ; because a hash collision occurs...
                case(next)
                case(label3)
                case(label4)
                case(label5)
                case(next)
                case(next)

label1:         ifskel(or,list1)    ; if first arg is the 'or' functor look at list1
                goto(next)              ;  otherwise go to the general section labelled 'next'

label2:         ifatom(trace,clause3)   ; if first arg is the 'trace' atom, execute clause 3
                goto(next)              ;  otherwise go to the general section labelled 'next'

label3:         ifskel(read,clause5)    ; if first arg is the 'read' functor, execute clause 5
                goto(next)              ;  otherwise go to the general section labelled 'next'

label4:         ifatom(notrace,clause4); (hash collision) if first arg is the 'notrace' atom, execute clause 4
                ifskel(write,clause6)   ;  otherwise if first arg is the 'write' functor, execute clause 6
                goto(next)              ;  otherwise go to the general section labelled 'next'

label5:         ifatom(nl,clause7)      ; if first arg is the 'nl' atom, execute clause 7
                goto(next)              ; otherwise go to the general section labelled 'next'

list1:          try(clause1)
                try(clause2)
                goto(next)              ; otherwise go to the general section labelled 'next'

;reference code for the first special section - try each clause sequentially or fail to 'next'
ref1:           tryskel(or,clause1)
                tryskel(or,clause2)
                tryatom(trace,clause3)
                tryatom(notrace,clause4)
                tryskel(read,clause5)
                tryskel(write,clause6)
                tryatom(nl,clause7)
                endssect                ;end of special section

GSECT1:next:    gsect                   ;start of general section
                try(clause8)            ; just one clause with a variable as first argument

SSECT2:         ssectlast(ref2)         ;start of special section; ref2 points to the reference code
                                        ; no hashing done because fewer than five clauses in section

                ifskel(call,clause9)
                ifatom(true,clause10)
                ifatom(repeat,list2)
                goto(fail)              ; no general section follows, so on failure, fail the call

list2:          notlast                 ;prepare for a possible shallow fail
                try(clause11)
                trylast(clause12)

;reference code for the second and last special section - try each clause sequentially or fail.
ref2:           tryskel(call,clause9)
                tryatom(true,clause10)
                tryatom(repeat,clause11)
                trylastatom(repeat,clause12)
```

Figure 2.8: Indexing code for the procedure in Figure 2.7. Only the eighth clause has a variable first argument, so the code comprises three sections: a special section for the first seven clauses, a general section for the eighth clause and a special section for the remaining four clauses.

While Warren does not say so, it seems plausible that the hashing function uses the first argument's type, value and arity if appropriate. The hash table is followed by code for five active hash codes. One hash code (`label1`) maps to the first two clauses, having `or/2` as a first argument. The hash codes of the first arguments of clauses 4 and 6, (`notrace` and `write/1` respectively), collide at `label4`, and the code that follows resolves the collision. The reference part of the first special section follows, starting at label `ref1`.

The general section is the indexing code for the eighth clause, as its first argument is a variable.

The second special section differs in layout from the first special section in that it doesn't include a hash table, as the number of clauses is less than five. A further difference arises because it is the last section; failure to select one of its clauses results in failure of the entire call. As a result, the ending `try`-type instructions are replaced by `tryLast`-type instructions.

2.11 Garbage Collection

Garbage collection is implemented in the global stack of the PLM. A relatively conventional mark-sweep system is used. The mark phase consists of identifying and marking all pieces of data that are part of the program's context—all clause instances and everything they can reach. The frames of all clause instances that are still 'live' in the stacks are marked, as are all the items that are referenced from variables within the frames. Frames are live if they can be continued into or if they can be backtracked into. Marking is done in two stages to avoid unnecessary recursion to dereference long chains. First, all cells in all live frames are marked but not dereferenced. Then, every reference from every cell is dereferenced and marked until the end of the defererence chain is reached or until an already-marked cell is encountered. The already-marked cell might have been marked as part of a frame marked

in the first marking stage.

Once the reachable memory locations have been marked, the sweep phase can begin. The idea is that all the reachable memory locations are 'swept', i.e. moved, to one end of the memory space, closing up all the unused gaps. Before this can be done, however, the values of the pointers in variables, references and structures are remapped—updated to point to the new locations of their referents. To accomplish this, once marking is finished, the displacement of every frame is calculated and stored in a special cell associated with the frame. Once this has been done for every frame, every variable, reference and structure frame pointer is dereferenced and the displacement associated with its referent's frame added to it.

When all the references are remapped, the memory locations are swept to the bottom of the global stack space.

2.12 Last Call Optimisation

The PLM is designed to be able to recover memory after a clause has completed determinately. Thus, for example, when the `foot` instruction executes, that is, after the clause is executed, if the most recent choice point is older than the current goal, the local frame associated with the current goal is recovered.

Subsequent to the design of the PLM, Warren discussed Tail Recursion Optimisation (TRO) [74, 76]. The idea is that it should be possible to recover the local frame of a determinate procedure just *before* the last call is made, because when the last call terminates, it would return to its caller only to have its caller immediately return to *its* caller. If the last call could be made return to its caller's caller, there would be no need to retain the caller's frame, since it would never be used again. In fact, the caller's frame could be deallocated before that last call is made, and the space freed up could

be used by the new clause instance. The scheme outlined applies when *any* last call is determinate, not just where the last call is recursive, so it is more generally called Last Call Optimisation (LCO). If the last call is recursive as well as determinate, the amount of space deallocated will be exactly the same as the amount of space needed by the new clause instance, thus leading to a constant rather than an increasing stack-space requirement.

The problem with this idea is that if any of the arguments of the last call include local variables, the space they occupy will be deallocated during the last call and accessed after the call, i.e. local variables in the last call will become *dandling references*. To solve this problem, Warren introduced *argument registers*, prefiguring their introduction into the later Warren Abstract Machine.[8] In this later version of DEC-10 Prolog (dubbed *Prolog-10* by Evan Tick [69]), arguments to a call are copied into argument registers $A_1..A_n$. This means that arguments to the last call *can* be local variables, so long as those variables do not themselves refer to the local stack frame. The only category of local variables that could be *unsafe* in this way are local variables that do not occur in the head of the clause. The solution adopted was to force such variables to be global at compile time.

2.13 Mode Declarations

DEC-10 Prolog offers the user the option of making *mode declarations*. These pragma-like declarations are used to reduce the amount of memory and the number of instructions necessary to implement the code. The user specifies whether each argument is 'input', 'output' or 'don't know'.

Consider the clause:

```
member(X,[X|R]).
```

[8]The WAM is discussed in Chapter 3, starting on page 53 of this thesis.

As it stands, the variables `X` and `R` must be declared to be global variables, since a list structure will be constructed. However, if the second argument is specified to be 'input' using a mode declaration, then the function of the list `[X|R]` is only to unify with the pre-existing list that is guaranteed to be the second argument. The list and the variables `X` and `R` are needed only to check the incoming structure and, in the case of `X`, to carry a value between the first argument of the clause and the first element of the incoming list. Considerable savings can be made therefore: the structure need not be permanent, saving memory, and the variables can be local rather than global, facilitating quicker recovery of the memory space used. Indeed, in this example, `R` can be implemented with a `void`, saving execution time as well as memory.

2.14 Discussion

DEC-10 Prolog was very successful, and for many years it was far and away the fastest and most reliable implementation of Prolog available. It was comparable in speed to many mature Lisp implementations of the day [72].

The PLM was designed as a 'staging post' between Prolog source code and DEC-10 Macro Assembler, and hence DEC-10 machine code. The output of the compiler was DEC-10 machine code and data for each clause, corresponding to the skeleton literals that comprised the structured arguments of the clause. In addition, machine code was generated for clause selection and indexing. From the description of the compiler, it appears that the clause itself was not stored, so that decompilation of the code to reconstruct the original clause would have been impossible.

The support of *dynamic code*—Prolog clauses that may be added or removed from the program at run time—is not addressed in the PLM. Dynamic code affects indexing methods and garbage collection in particular, and of

course the semantics of `assert` and `retract` predicates would have to be clearly defined.

Perhaps the most intriguing question is how much of its outstanding performance DEC-10 Prolog owes to its tight integration with the slightly unorthodox architecture of the DEC-10, bearing in mind that the DEC-10's automatic dereferencing property is explicitly used in the design (see page 30).

It would be also be interesting to know whether the extra resources devoted to the special treatment of first level structures—the registers `B` and `Y` and the level 1 instructions—and the combining of some frequently-occurring instruction sequences (such as combining `neck` and `foot` to form `neckFoot`) were warranted by the improvement in performance observed.

Open Prolog borrows a good deal from DEC-10 Prolog. Like DEC-10 Prolog, Open Prolog is based on structure sharing and is stack based rather than register based, using global, local and trail stacks in very similar ways. Variables are analysed in almost the same way, and some instructions and registers have the same names in both implementations. The garbage collection algorithm used on the global stack in Open Prolog is very similar to that used in the PLM. The PLM is, however, an intermediate target for the compiler; as noted above, the final output of the compiler is DEC-10 machine code. The Open Prolog compiler, by contrast, produces code for an abstract machine which supports dynamic code, code inspection, Last Call Optimisation and catch-and-throw error handling.

Chapter 3

The Warren Abstract Machine

3.1 Introduction

In 1983, David H.D. Warren published a description of what has come to be known as the *Warren Abstract Machine* [75], more succinctly known simply as the *WAM*.[1] The WAM and the PLM are similar in many ways; in both machines, Prolog source code is compiled into instructions for an abstract machine; both have two main stack areas and a trail; both use tag-and-data representations of literals. The biggest differences are:

- Structure copying is used instead of structure sharing to represent compound terms;
- The WAM is a register-based machine; the PLM is stack-based. The WAM uses *argument registers* and *temporary registers* to hold Prolog literals. The PLM has no such data registers.

These differences are very far-reaching in their effects, and for all their similarities, the WAM and the PLM are very different in their operational char-

[1] See [1] for a tutorial-style introduction to the WAM. For an overview of the PLM and the WAM, see [38].

acteristics. This section gives a brief summary of the WAM.

3.2 Overview

Source code is compiled into the machine code of the WAM. The machine code is classified as consisting of:

- *get* instructions, corresponding to the arguments in the head of the clause;
- *call* instructions, corresponding to goals in the body of the clause, each preceded by
- *put* instructions, corresponding to the arguments of the goal.

In addition, *unify* instructions correspond to the arguments of a structure or list. Warren also includes the classification of *procedural* instructions, such as the `call` instruction, and *indexing* instructions, used to select clauses.

The `unify` instructions correspond to the arguments of a structure or list. If a goal argument is to be unified with a structure whose arguments are represented by these instructions, then two possibilities exist:

- If the goal argument is already a structure, then the unify instructions simply check that the structure and its arguments match.
- If the goal argument is an uninstantiated variable, then the unify instructions construct a copy of the structure they represent and assign it to the variable.

The WAM addresses the two possibilities using two *modes*. In the first case, the machine is placed in the *read mode*, and in the second case, it is placed in the *write mode*.

3.3 Variables and Data Areas

As with the PLM, a great deal of attention is devoted to reducing memory usage. The two main differences between the PLM and the WAM come into play here; the WAM uses structure copying and is register based whereas the PLM uses structure sharing and is stack-oriented.

Because the WAM uses structure copying, structures are copied as needed into a special data area called the *heap*. If a structure contains an uninstantiated variable when copied, space is allocated in the structure's representation on the heap for the value of the variable. If the variable is instantiated later, its new value is stored in that space. In this discussion, let us call these variables *structure variables*.

The other kind of *source variables*—variables that appear in Prolog source code—are variables that appear as arguments in the head or in goals. Let us call these arguments *argument variables*. Argument variables are assigned to *argument registers* in the WAM; before a goal is called, the appropriate arguments are loaded into argument registers. There is a clean distinction between structure variables and argument variables: structure variables are assigned space *in situ* in the structure in the heap; argument variables are assigned into argument registers. A source variable that is both a structure variable and an argument variable gets space in the structure in the heap and an argument register at the appropriate time; the argument register will contain a reference to the variable in the structure. For example, we can classify the variables in the clause `member(X,[Y|Z]) :- member(X,Z).` as follows:

- The source variable `X` is an argument variable in the head and in the goal.
- The source variable `Y` is a structure variable only.

- The source variable `Z` is a structure variable in the head and an argument variable in the body.

As we have seen, the WAM dispenses with the PLM's global frame by allocating storage for structure variables in the heap. Likewise, the use of argument registers dispenses with the need for the PLM's local variables. However, argument variables must be stored in memory under some circumstances. Consider the variable `S` in the clause:

```
quicksort([K|T],S) :-
        partition(T,K,L,G),
        quicksort(L,S1),
        quicksort(G,S2),
        append(S1,[K|S2],S).
```

The value of the variable `S` that is present when the clause is called must be stored in memory so that when the intervening calls— to `partition/4` and to `quicksort/2`—have concluded, it can be loaded into an argument register prior to the call to `append/3`.

Variables that do not need to be stored in memory are called *temporary variables* in the WAM. A temporary variable has its first occurrence in the head or in a structure or in the last goal, and does not occur in more than one goal (where, for this purpose, the head of the clause and the first goal are taken as one goal).

Argument variables that must be stored in memory (i.e. argument variables that are not classifiable as temporary variables) are called *permanent variables* in the WAM and are stored in the clause instance's *environment* in a data area called the *stack*, along with the control information.

Similarly to the earlier DEC-10 Prolog implementation, where a local variable could be unsafe (see page 50), a permanent variable can be *unsafe*. A permanent variable that references an unbound permanent variable at the time it is deallocated will leave a dangling reference, and deallocation can

occur due to environment trimming or to the deallocation of all permanent variables prior to a last call. The only permanent variables that could do this are variables that do not occur in the head of the clause or in an inner literal of a structure. Unsafe variables are detected at compilation time and are allocated space on the heap if necessary at run time to avoid this possibility.

Finally, in his description of the WAM, Warren sometimes refers to permanent variables as *stack variables* or *local variables*, and similarly he sometimes refers to variables placed in the heap as *global variables*.

Backtracking presents a difficulty for argument variables stored only in argument registers. When a goal is called, the appropriate arguments are copied into argument registers. If backtracking subsequently attempts to resatisfy the goal using a different clause, these arguments must be reloaded into the argument registers. To facilitate this, the WAM uses a *choice point* data structure, which is constructed when needed on the stack. The choice point stores the goal arguments in memory so that the argument registers can be restored on backtracking.

As with the PLM, the analysis of variables takes into account whether a particular use is the first use or a subsequent use. In addition, if the variable is classified as permanent, account is taken of its *last use*. As soon as a permanent variable has been used for the last time, the space it occupies in the environment could potentially be recovered by 'trimming' it from the environment, described later.

3.4 Data Representation

Warren envisaged the WAM being implemented in a 32-bit machine, and suggested a 'provisional' tag-and-data format for data items, shown in Figure 3.1. Lists have their own distinct tag, and floating point numbers are handled as variants of structures.

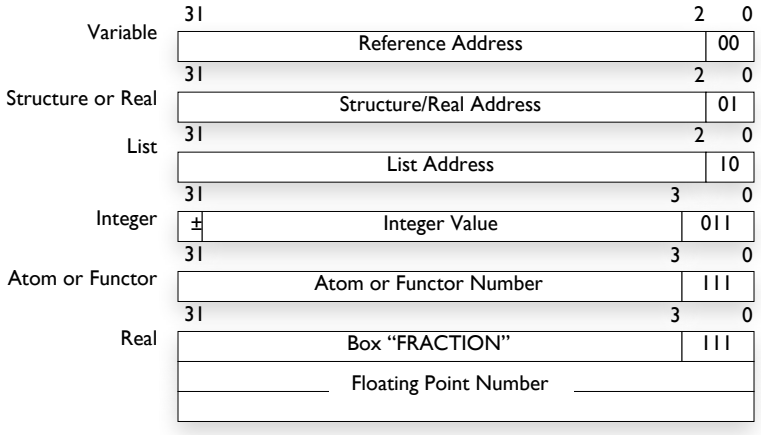

Figure 3.1: Suggested Data Formats for the WAM. An unbound variable is represented as a reference to itself.

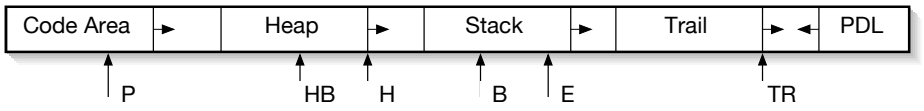

Figure 3.2: Layout of the WAM's memory areas. When unifying two variables, the higher addressed location is assigned a pointer to the lower addressed cell. This, combined with the low-to-high ordering of the heap and stack, and the fact that both grow upwards, is sufficient to prevent dangling references.

3.5 Data Areas and Structures

The main data areas (see Figure 3.2) are the code area, the heap, the stack and the trail, with a small push-down-list space used during unification. As with the PLM, Warren doesn't describe how the textual names of atoms and functors are represented, as it is not important to the overall function of the machine. The code area contains instructions and other data representing the program itself.

The heap is used to store newly constructed structures and lists. It may

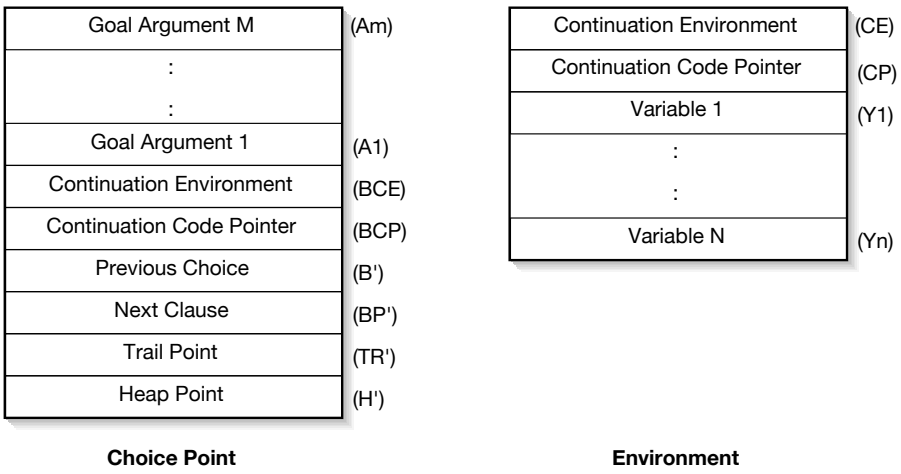

Figure 3.3: Layout of WAM Choice Point and Environment data structures.

also store what Warren calls *global* or *heap variables*. These are variables whose lifetime exceeds the lifetime of a clause instance's environment, but which do not form part of a structure.

The stack is used to contain environments and choice points, the format of which is shown in Figure 3.3.

3.6 Registers

Warren details nine special purpose registers and an indefinite number of argument registers and registers for temporary variables, as listed in Table 3.1

The Argument Registers (the `An` registers) and the Temporary Variable Registers (the `Xn` registers) are the same registers (i.e. `Ai` is the same register as `Xi`), named differently to reflect their different usages. The WAM makes use of this fact to minimise data movements within the register set.

Name	*Description*
`P`	Program Counter (points into the Code Area)
`CP`	Continuation Program Counter (points into the Code Area)
`E`	Pointer to the last Environment (on the Stack)
`B`	Pointer to the last Choice Point (on the Stack)
`A`	Pointer to top of Stack
`TR`	Pointer to top of Trail
`H`	Pointer to top of Heap; used, e.g., when writing structures into the Heap
`HB`	Pointer to Heap Backtrack Point, i.e. the top of the Heap if execution had backtracked to `B`
`S`	Structure Pointer, points to a structure in the Heap; used by `unify...` instructions in read mode)
`A1`,`A2`,...	Argument Registers
`X1`,`X2`,...	Registers for Temporary Variables

Table 3.1: WAM Registers.

3.7 Instruction Set

In his report, Warren classifies the WAM's instruction set into *get* instructions, *put* instructions, *unify* instructions, *procedural* instructions and *indexing* instructions. Table 3.2 lists all the instructions.

3.8 Program Representation

Each clause in a Prolog program is separately encoded. Clauses having the name name and arity, i.e. clauses belonging to the same procedure, are linked together using indexing code. The general scheme for encoding clauses is described in this section, and indexing is described later.

Get Instructions	
`get_variable Yn,Ai`	`get_variable Xn,Ai`
`get_value Yn,Ai`	`get_value Xn,Ai`
`get_constant C,Ai`	`get_nil Ai`
`get_structure F,Ai`	`get_list Ai`
Put Instructions	
`put_variable Yn,Ai`	`put_variable Xn,Ai`
`put_value Yn,Ai`	`put_value Xn,Ai`
`put_unsafe_value Yn,Ai`	
`put_constant C,Ai`	`put_nil Ai`
`put_structure F,Ai`	`put_list Ai`
Unify Instructions	
`unify_void N`	
`unify_variable Yn`	`unify_variable Xn`
`unify_value Yn`	`unify_value Xn`
`unify_local_value Yn`	`unify_local_value Xn`
`unify_constant C`	`unify_nil`
Procedural Instructions	
`proceed`	`allocate`
`execute P`	`deallocate`
`call P,N`	
Indexing Instructions	
`try_me_else L`	`try L`
`retry_me_else L`	`retry L`
`trust_me_else fail`	`trust L`
`switch_on_term Lv,Lc,Ll,Ls`	
`switch_on_constant M,Table`	
`switch_on_structure M,Table`	

Table 3.2: The WAM Instruction Set. The abbreviations `Xn`, `Yn`, `C` and `F` represent, respectively, a temporary variable, a permanent variable, a constant and a functor. `P` is the address of a procedure and `N` is the number of variables. `L`, `Lv`, `Lc`, `Ll` and `Ls` are addresses of clauses or clause sequences, and `Table` is a hash table of size `M`.

`P`	`P :- Q`	`P :- Q,R,S.`

get args of `P`
`proceed`
	get args of `P`
	put args of `Q`
	`execute Q`
		`allocate`
		get args of `P`
		put args of `Q`
		`call Q,N`
		put args of `R`
		`call R,N1`
		put args of `S`
		`deallocate`
		`execute S`

Figure 3.4: WAM Encoding Schemas for three different types of clauses.

3.8.1 General Clause Schemas

First, the general scheme for encoding clauses is illustrated in Figure 3.4. In the first case, a unit clause, the clause code simply *gets* the arguments of the clause instance; the `proceed` instruction terminates the clause by returning program execution to the location pointed to by `CP`. In the second example—a clause having just one goal in the body—an environment is definitely not needed, so the clause code consists of instructions to *get* the arguments of the clause instance, to *put* the arguments of the single goal and then to transfer execution to that procedure using the `execute` instruction. In the third example, an environment is needed because there is more than one goal in the body. Thus the first instruction is to allocate the environment, and the second-last instruction is to deallocate it.

One other thing worth noting here is that the `call` instruction takes two arguments. The first argument is the address of the called goal's first clause or index code, as might be expected. The second argument, however, is used to help 'trim' the environment: it represents the number of variables that must be retained in the current environment. At the point when a call is made, certain permanent variables may no longer be needed, and if they are

at the edge of the environment data structure (see Figure 3.3), they can be trimmed off. If the environment happens to be at the top of the stack, the space trimmed in this way is immediately reusable; thus it can be viewed as a generalisation of last call optimisation.

3.8.2 `Get` Instructions

The *get* instructions encode the arguments of the head of a clause and are executed just after a clause instance has been invoked. At that time, the arguments being passed to the clause instance have already been placed in the argument registers `A1`, `A2`, `...` , and the function of the instructions is to unify the head arguments they represent with the arguments in the goal. For example, the instruction `get_constant C,Ai` represents the i^{th} head argument, which is the constant `C`. When executed, it unifies the constant with the i^{th} incoming goal argument, which is in argument register `Ai`. The instruction `get_nil Ai` is a special case of this instruction, where the constant is the empty list `[]`.

Instructions representing variable head arguments come in two varieties: `get_variable Vn,Ai` for the first occurrence of a variable and `get_value Vn,Ai` for subsequent occurrences of a variable. The difference between the two is very significant; in the first case, it is known that `Vn` is unbound. As a runtime optimisation, such variables are not explicitly initialised; instead, when first encountered, `Vi` is simply assigned the value contained in `Ai`. In the second case, it is known that `Vi` contains a value, so the contents of the two registers are unified and dereferenced, a considerably more complex operation than simple assignment.

The remaining two *get* instructions are related; `get_structure F,Ai` and `get_list Ai` represent, respectively, the start of a structure or list as a head argument. Both instructions are responsible for setting the WAM up for

unifications of the structure or list's subterms. Given that the WAM is a structure copying implementation, these instructions may cause the WAM to start constructing a new structure or list on the heap. The instructions dereference the incoming argument `Ai` and operate as follows:

- If it is a variable, then a new structure or list must be constructed, or 'written' to the heap. Hence, the WAM is put into the 'write' mode, which affects the operation of the `unify...` instructions.
- If it is not a variable, if the result is either a structure with a matching principal functor or a list, as appropriate, then the existing subterms of the head and goal arguments must be checked, and the WAM is put in the 'read' mode, again affecting the operation of the `unify...` instructions. (If the functors don't match or the item is not a list, execution fails.)

3.8.3 `Unify` Instructions

The `unify...` instructions represent structure and list arguments, and their operation is affected by whether the WAM is in the read mode or the write mode. Depending on which mode these instructions are executed in, they make use of two of the WAM's special purpose registers, the `S` (for *structure*) register or the `H` (for *heap*) register. For example, if the instruction `unify_constant C` is executed in read mode, the instruction unifies the corresponding structure argument, pointed to by register `S`, with the constant `C`. In the write mode, however, the instruction writes the constant `C` onto the heap, as pointed to by register `H`.

Instructions representing variables in a structure come in four varieties: `unify_variable Vn`, `unify_value Vn`, `unify_void N` and `unify_local_value N`. Of these, `unify_variable Vn` and `unify_value Vn` work similarly enough to `get_variable Vn,Ai` and `get_value Vn,Ai` but

with the difference that in the write mode, something has to be written to the heap to be part of the structure being created. With `unify_variable`, a new structure variable is created on the heap and a reference to it placed in `Vn`. With `unify_value` the situation is a little more complex. The simplest thing would be to copy the contents of `Vn` into the heap, but the problem is that this might result in a dangling reference. If `Vn` can not be guaranteed to contain either a constant or a reference to another structure variable, it could contain a reference to a permanent variable, i.e. a stack variable, which could be deallocated before the structure that is being constructed is discarded. Consequently, the `unify_value Vn` instruction is only used where the compiler can guarantee that the contents of `Vn` will be 'safe' in this sense.

Where the guarantee of safety can not be given, the instruction `unify_local_value Vn` instruction is used instead. This instruction performs a check to ensure that `Vn` does not reference a stack variable; if it does, a new unbound structure variable is written to the heap, and the stack variable in question is set to reference the new variable. (See also the instruction `put_unsafe_value Yn,Ai` below, where a similar situation may arise.)

The `unify_void N` instruction is used where one or more void variables occur in a structure. There is no corresponding instruction `get_void Ai` because the argument `Ai` would never be used, so there is no need to examine it. Analogously, the read mode, `unify_void N` just skips `N` arguments, but in the write mode, `N` unbound variable are initialised in the heap as part of the structure being created.

3.8.4 `Put` Instructions

`Put` instructions are the third category of instructions corresponding to arguments in the WAM. As their names might suggest, these instructions are concerned with putting arguments into registers, either argument registers

or temporary variable registers. For example, the instruction `put_const C,Ai` puts the constant `C` into register `Ai` (equivalently, register `Xi`). The instruction `put_nil Ai` is a specialised version of this instruction, where the constant is the empty list: `[]`.

Again, there are four instructions for representing variables. First, `put_variable Yn,Ai` initialises `Yn` and puts a reference to it into `Ai`. The instruction `put_variable Xn,Ai` is used in the final goal of a clause, where the environment—if it existed—is gone, and all that remains are temporary variables and argument registers. The problem is that the registers holding temporary variables will no longer be available as soon as the `execute` instruction is executed. Therefore, this instruction initialises a new variable on the heap, and places a reference to it in `Xn` and `Ai`. Next, the instruction `put_value Vn,Ai` simply copies the value of `Vn` into `Ai`. Finally, the `put_unsafe_value Yn,Ai` instruction encodes the last occurrence of a variable that may still reference a stack variable. If so, a new heap variable is created and the register `Ai` is set to reference it.

3.8.5 Nested Structures

The WAM instruction set does not support nested structures or sublists directly. Instead, where a nested structure or sublist is required, a temporary variable is substituted for it, and the temporary variable itself is unified with the nested structure or list. If the nested structure or sublist is part of the head, and is therefore to be unified with terms being received from the caller, the instructions to unify the stand-in variable with the nested item *follow* the sequence of unify instructions in which it appears. If the nested structure of sublist is part of the body, and is therefore part of an argument to be passed to a subgoal, the instructions to unify the nested item with its stand-in variable *precede* the sequence of instructions in which the variable

```
concatenate([],L,L).
concatenate([X|L1],L2,[X|L3]) :- concatenate(L1,L2,L3).

concatenate/3: switch_on_term C1a,C1,C2,fail.

C1a:    try_me_else C2a          % concatenate(
C1:     get_nil A1               %    []
        get_value A2,A3          %    L,L
        proceed                  % ).

C2a:    trust_me_else fail       % concatenate(
C2:     get_list A1              %    [
        unify_variable X4        %     X|
        unify_variable A1        %     L1],L2
        get_list A3              %    [
        unify_value X4           %     X|
        unify_variable A3        %     L3]) :-
        execute concatenate/3    % concatenate(L1,L2,L3).
```

Figure 3.5: WAM Encoding of `concatenate/3`, from [75].

appears. Thus, nested structures give rise to chains of `get...` or `put...` instructions, 'connected' via temporary variables.

3.9 Examples

Figure 3.5 lists the WAM code for the `concatenate/3` predicate, and Figure 3.6 is an example of the encoding of nested structures.

The `concatenate/3` example highlights many of the advantages the WAM has over the PLM. Perhaps the most significant advantage is this: if the call to `concatenate/3` is determinate on the first argument, no choice point is allocated, and, because there is only one goal in the clause body, the WAM

```
d(U*V,X,(DU*V)+(U*DV)) :- d(U,X,DU),d(V,X,DV).
        try_me_else...              % d(
        allocate
        get_structure '*'/2,A1      %    *(
        unify_variable A1           %      U,
        unify_variable Y1           %      V),
        get_variable Y2,A2          %    X,
        get_structure '+'/2,A3      %    +(
        unify_variable X4           %      SS1,
        unify_variable X5           %      SS2),
        get_structure '*'/2,X4      % SS1 = *(
        unify_variable A3           %         DU
        unify_value Y1              %         V),
        get_structure '*'/2,X5      % SS2 = *(
        unify_value A1              %         U,
        unify_variable Y3           %         DV)) :-
        call d/3,3                  % d(U,X,DU),
        put_value Y1,A1             % d(V,
        put_value Y2,A2             %   X,
        put_value Y3,A3             %   DV
        deallocate
        execute d/3                 % ).
```

Figure 3.6: WAM Encoding of `d/3`, with the addition of the `allocate` and `deallocate` instructions inadvertently omitted from [75]. This is an encoding of a differentiation rule in Prolog, from [72].

completely avoids having to construct an environment.[2] The second noticeable advantage is that pass-through variables, such as the second argument in the second clause of the predicate, incur no run time overhead; notice that the second argument, in register `A2`, is not manipulated at all in the code beginning at label `C2`.

The second example highlights how the WAM deals with nested struc-

[2]The PLM's local stack frame encompasses the functionality of both the WAM's environment and choice point data structures.

tures. The code for a nested structure follows the structure's first appearance in a head argument, and precedes the structure's first appearance in a body argument; in each case, the structure's place is taken by a stand-in variable—a temporary register. By comparison with the PLM, the WAM is at a disadvantage if large amounts of deeply nested structures are to be processed, as it will have to execute code to construct new copies of these structures, whereas the PLM will simply have to make a reference to the existing term.

3.10 Indexing

The indexing scheme is similar to that of the PLM in many ways. As with the PLM, clauses are indexed by their first arguments, clauses are grouped into sections, hashing may be used to select viable clauses within a section, and finally sequential selection if more than one viable clause results. The details are somewhat different, however.

The sequence of clauses in a procedure is partitioned into sections corresponding exactly to the PLM's general and special sections (see page 44) each of which contains either one clause whose first argument is a variable or a maximal subsequence of consecutive clauses whose first argument is a non-variable. For example, the `call/1` procedure used as an example for the PLM (page 46) is divided into three clause blocks, S1 to S3:

$S1$
```
call(X or Y) :- call(X).
call(X or Y) :- call(Y).
call(trace) :- trace.
call(notrace) :- notrace.
call(read(X)) :- read(X).
call(write(X)) :- write(X).
call(nl) :- nl.
```

$S2$
```
call(X) :- ext(X).
```

$S3$
```
call(call(X)) :- call(X).
call(true).
call(repeat).
call(repeat) :- call(repeat).
```

Each clause block may be visited in sequence during clause selection using `try_me_else`, `retry_me_else` and `trust_me` WAM instructions. For example, for the above procedure, the code would be:

```
try_me_else(S2)
    Code for Major Segment S1
S2: retry_me_else(S3)
    Code for Major Segment S2
S3: trust_me
    Code for Major Segment S3
```

Viable Clauses

Once a section has been selected, (and if it is not a single clause with a variable as its first argument), a `switch_on_term` instruction selects one of four outcomes depending on whether the first argument of the goal is a *variable*, a *constant*, a *list* or a *structure*.

If the first argument is a variable, all clauses in the section could potentially match so control is transferred to a sequence of `try_me`, ..., `retry_me`, ..., `trust_me` instructions to allow each clause in the block to be visited in sequence.

If the first argument is a constant or a structure, a further selection instruction is executed: a `switch_on_constant` or `switch_on_structure` instruction selects a sequence of clauses on the basis of the actual value of the first argument. The selection may be done using hash tables.

For non-variables (lists, constants or structures) at this point, three outcomes are possible: first, if there is no match, execution fails in this clause block; second, if exactly one clause matches, it becomes the only clause in the clause block that is visited; third, if a number of clauses match, each is visited using the familiar `try_me`, ..., `retry_me`, ..., `trust_me` chain of instructions.

Overall, therefore, clause indexing comprises up to three stages: selection of a clause block (omitted if there is only one clause block), selection of viable clauses within the clause block (omitted if there is only one clause in the block), and finally, selection of a clause from the viable clauses (omitted if there is only one viable clause).

3.11 Discussion

The WAM represents a considerable advance over the PLM. The use of argument and temporary registers, the splitting of the environment frame from the choice point frame (versus the local stack frame of the PLM) are notable. The elimination of the PLM's structure skeletons by compiling them into `unify...` instructions is another notable change, as is the attendant elimination of the need for the rather awkward *molecule* needed in the PLM to represent a constructed structure. In this writer's view, two features stand

out. These are the potential to avoid completely the generation of an environment or choice point and the ability to avoid the generation of instructions for arguments that are present but unused. Both these features come into play in the `concatenate/3` predicate (Figure 3.5). Because the clause consists of only one goal, no environment needs to be generated; if the call is determinate, no choice point will be generated either. The second argument, `L2`, occupies argument register `A2` and remains there throughout execution of the clause; no instructions are generated for it.

With respect to the aims of this thesis, however, the WAM presents some difficulties, principally with regard to decompilation. When a clause is compiled to WAM code, what is produced is a sequence of instructions. In particular, neither structures nor variables are retained explicitly. In place of the structures present in the source code, instructions are generated which construct or test for the presence of these structures at runtime. It would be necessary to 'play back', i.e. symbolically execute, these instructions to reconstruct the structures.

Similarly, variables in the clause give rise to instructions that manipulate argument registers, temporary registers, stack variables and heap variables. To reconstruct the variables, it would be necessary to perform, effectively, a liveness analysis of all the registers used by the clause code, beginning with the argument registers. The presence of void, unused and anonymous variables would have to be inferred.

While the two problems referred to above are not insurmountable, they are reasonably complex and would represent quite a significant effort. See [15] for another view of this.

Similarly to the PLM, the support of *dynamic code*—Prolog clauses that may be added or removed from the program at run time—is largely omitted from the WAM.

Chapter 4

Open Prolog

4.1 Summary

In this chapter, the motivation behind the design is discussed and the design choices are set out. Contributions covered in this chapter include all major aspects of the design—the overall design philosophy, the use of two PCs while performing unification, the selection of structure sharing.

4.2 Design Considerations

As stated in the introduction, the aim of the work described here was to design an implementation of Prolog that would combine the advantages of compiler-based implementations—speed and memory efficiency—with those of interpreter-based implementations—ease of inspection of programs, ease of modification and ease of debugging—in one mode of operation. In addition, of course, it was hoped that the resulting implementation would be compact[1] and relatively easy to implement.

[1]Or at least 'compact' in the sense of an economy of concepts.

4.2.1 Overall Design

Consider an informal implementation 'schema' where source code is translated into an intermediate form which is interpreted on the host machine. There would be a 'translation gap' between the source code and the intermediate form and an 'interpretation gap' between the intermediate form and the host processor. The translation gap would be absent in the case of a pure interpreter, and the interpretation gap would be absent in the case of a host-code-producing compiler. It seems that every high-level language implementation could be mapped onto this schema.

An implementation with some element of compilation will, in principle, always be better than a pure interpreter because the compiler will be able to take advantage of information available in the source code to produce an intermediate code that will consume less resources at runtime. For example, in the PLM and the WAM, variable analysis carried out at the time of compilation identifies variables that need not be allocated any memory space, saving memory. The same analysis performed at runtime—where possible—will of necessity take execution time. It is apparent, therefore, that our design must incorporate some level of compilation, so, in the schema above, the translation gap will be bridged by some form of compilation. Aggressive compilation techniques, on the other hand, while improving execution and memory performance, increase compilation time and typically produce output code that can not be related back to the source code. In these cases, it is impossible to inspect or modify the compiled Prolog code. Of course, the code could be annotated with information to enable sections of the code to be related back to sections of the source code. This is not a completely satisfactory solution for Prolog however, because if a source file is consulted, modified and then reconsulted, code from the original consultation could still be active, while its source text could be gone from the file.

Thus, it seems that the single mode of operation desired for the imple-

mentation should incorporate just enough compilation to give it high performance, but not enough to destroy the correspondence between source and intermediate code nor to unduly delay clause assertion.

Ideal Architectures and particularly *Direct Correspondence Architectures (DCAs)* offer a coherent framework within which to consider these and related issues.

An Ideal Computer Architecture[2] is an architecture, specific to a particular high level language, which exhibits a one-to-one correspondence between the high level language representation of a program and the associated machine coding. It is also designed to minimise instruction size, execution time and compilation time. A Direct Correspondence Architecture is an implementable architecture that approaches the ideal by adhering to the principles of Ideal Architectures.

In the Ideal Architecture computing model, computation is considered a two phase process, translation and interpretation. Translation is the process of changing or translating the description of a task in a high level language into an intermediate form. The intermediate form is a sequence of instructions where an instruction is a function that changes execution state. Interpretation is the act of evaluating the instructions to effect execution. It is performed by an image machine. The image machine corresponds to the image of the machine that translation from the source is targeted to, and the set of instructions of the image machine are the execution architecture. The individual instructions are called the image instructions. The image machine is implemented on a host processor using an emulator.

A requirement of Ideal Architectures and DCAs is that there should be a one-to-one relationship between the image architecture and the high-level-language encoding. This means that there should be a one-to-one correspondence between the source code and the image code representation, but also

[2]This account of Ideal Computer Architectures is based on [37] and [77].

that there be a direct correspondence between the state of the image machine as it performs the image program, and the state of the source program as it would be performed according to the language's operational semantics.

These principles offer an attractive basis for partitioning the design of the implementation into the translation and execution phases and were broadly adopted in the design of Open Prolog. As a consequence, the design incorporates compilation techniques to bridge the translation gap and an emulator to bridge the interpretation gap. In the middle is an abstract machine whose instructions are emulated for execution and decompiled to provide code inspection facilities.

4.2.2 Clause Representation

Perhaps the first thing to realise is that a clause is both program *and* data. Clearly part of a program, a clause is also data for, say, a program that lists clauses. Thus, its representation must facilitate execution as part of a program and identification as a piece of data. If it is represented simply by the code needed to fulfil its function when executed, it would not be possible to list or inspect it. It was therefore decided to represent code using a tagged representation scheme, with a tagged representation for every component of the clause, including arguments, data structures, the neck, the foot, control instructions and the cut.

To incorporate information gleaned from compilation, the instruction set was enriched to represent different types and usages of variables, to specially signify the last subterm in a structure and to amalgamate certain frequently occurring sequences of instructions, e.g. a clause body consisting of a cut could give rise to just one instruction `neckCutFoot` rather that three `neck`, `cut` and `foot`. While this increases the size of the abstract machine's instruction set, the benefits in terms of performance and memory consumption

would surely be worthwhile. As far as decompiling the code back to source, the extra complexity is minimal—often a question of adding some extra entries to a table.

4.2.3 Implementation of Unification—Dual PCs

Unification generally occurs between a goal and a clause head. Each argument in turn is unified with its corresponding argument. In the PLM, the head terms are compiled into host machine code, and are responsible for fetching their corresponding goal arguments, which are treated as data. One could envisage a similar situation here: the emulator would take the instruction from each head term and execute it. This would locate and fetch its corresponding goal argument and process it. In most cases the goal term to be fetched by the next head term would be adjacent to the term just fetched. If the pointer was stored and left for the next head term, the overhead of locating the next goal term would be obviated. Thus it seems natural to reserve a special goal argument pointer for 'walking' the sequence of goal term instructions as well as the existing PC which likewise walks the head term instructions.

Given that there is little or no distinction between the codes used to represent terms in the head and those in the goal, it seems artificial to distinguish between the the 'PC'—i.e. the head argument pointer—and the goal argument pointer. Accordingly, it was thought appropriate to treat both pointers equally, resulting in a system that employs two program counters during unification. The PCs are termed the Head PC (`HPC`) and the Body PC (`BPC`).

4.2.4 Structure Sharing and Structure Copying

As discussed in Chapter 1 on page 19, there are two well-establised schemes for representing dynamically constructed data—structure sharing and structure copying. Without doubt, either technique could be applied to the implementation. For Open Prolog, structure sharing was chosen because of its affinity with the standard scheme for representing procedure- or clause-instances. Recall (see page 20) that a clause instance could be represented by a pair of pointers, one to the clause code and the other to a frame of information specific to the particular clause instance. In structure sharing, a constructed term is represented in a very similar way: a pointer to the structure's skeleton and another pointer to a frame of information specific to that particular term. By adopting structure sharing, it was hoped that the similarity between how clause instances and constructed terms are represented would result in a simpler abstract machine and a simpler overall implementation. The fact that clause instances can require two frame pointers and may hold various 'housekeeping' information does complicate matters, and it is moot whether structure sharing 'pays off' in that sense. Nevertheless, this was the reason for selecting structure sharing over structure copying. It would be interesting indeed to rebuild Open Prolog on a structure copying platform.

4.2.5 Dynamic Code Support

One of the aims for this design is to support the modification of a program by way of adding or removing clauses, that is, that it should offer dynamic code support. This means, of course, that procedures are no longer 'monolithic' entities that don't change once consulted. Thus:

- Compilation schemes that treat the whole procedure at once(e.g. see [70, 68]) will not be usable.

- Clause indexing schemes such as used in the PLM and WAM (described on pages 44 and 69 respectively), where the index code is constructed once at the time of compilation are likewise ruled out.

Clause indexing is a particularly serious issue, because apart from speeding up procedure execution, the information obtained at runtime through clause indexing can frequently make a call determinate, yielding very considerable savings in memory consumption.

One solution is to replace indexing with a lookahead feature, whereby when a clause has been selected for execution the machine looks ahead to see in an alternative clause might be available; if not, the selected clause is executed determinately. Unfortunately, this entails a further complication: an alternative might be retracted, giving, effectively, a 'dangling alternative reference'. Or, an alternative clause might be asserted after the lookahead process concluded that no alternative was available.

Over the years, many differing solutions have been adopted [52], but a paper by Linhholm and O'Keefe [49] showed that the *logical update view* proposed by Moss in [52] could be implemented efficiently. Thus, this design, being unable to use true indexing in the style of the WAM or PLM, implements the logical update view proposed by Moss.

At this point, the most significant features of the design are decided:

- A DCA-influenced design, intended in one mode of operation to facilitate efficient execution and straightforward compilation and decompilation,

- A uniform tag-and-data intermediate code with a one-to-one correspondence between source code and intermediate code elements, enhanced to include or encode compiler-derived optimisations.

- A dual-PC arrangement for implementing unification in an efficient manner while preserving the uniformity of the encoding scheme,

- The use of structure sharing to represent constructed terms, in the expectation that the similarities between clause instance and constructed term representation would lead to a simpler design and consequently, a more compact implementation.

- Support for dynamic clause code using clause lookahead rather than clause indexing, and using the logical update view of clause assertion and retraction.

The remainder of the chapter describes of how these decisions were carried over into the design.

4.3 Overview

Open Prolog is a structure-sharing stack-based implementation based on an abstract machine called the Open Prolog Abstract Machine (OPAM). The OPAM is designed to minimise the 'compilation gap' between source code and OPAM code, and an OPAM emulator is written in 68000 assembler and also partly in Pascal, C and Prolog for execution on a Macintosh. Assembler was chosen for full control over the exact mapping between the OPAM and the host processor. Pascal and C were used where such control was not critical. Many of the built-in predicates lent themselves naturally to implementation in Prolog, with the lowest level facilities being implemented in a procedural language.

When programs are consulted, OPAM code is produced for each clause, corresponding to (i) the arguments in the head of the clause, (ii) each call and its arguments in the body of each clause, (iii) certain features of the clause itself, (for instance, a `neck` instruction is produced to go between the code for the head and the code for the body), and (iv) the name of the clause—used to form its index code. As each clause is compiled, it is added to the

code base of the system and linked safely to index code for the procedure it belongs to.

OPAM code is both representational and executable. Each component of a clause, apart from comments and variable names, is represented by OPAM code, so that it may be decompiled directly to Prolog source. The code is executable on the OPAM. The OPAM itself is unusual in that it has two program counters—the *Head Program Counter (HPC)* and the *Body Program Counter (BPC)*—and operates in one of three modes: fetching from both PCs when unifying terms, and fetching from the HPC or the BPC when executing normally.

Clauses are linked by index code. This code allows a candidate and alternate clause to be identified by reference to the first argument of the goal, thus identifying the determinacy of calls. Allowance is also made for the currency of clauses on the basis of their assertion and retraction time stamps and on the time stamp of the program execution. Garbage collection operates on the data areas and on the code space. Built-in predicates are written in Prolog and in host code, and plug-in 'external' predicates can be added as Macintosh resources, communicating with Open Prolog through a parameter block-based interface. Included in the implementation is an integrated text editor, providing a complete development environment for the programmer. A rudimentary external events interrupt mechanism is also incorporated.

4.4 Variable Analysis

With one modification, OPAM follows the PLM in classifying variables as *global*, *local*, *temporary* or *void*.

- Variables that appear only once, and not in any structure, are *voids*.

The following rules then apply to non-voids:

- Variables that appear in a structure or as arguments of the last goal are classified as *global*.
- Variables occurring only in the head of a clause, and not in any structure, are classified as *temporary*.
- Variables occurring anywhere but in the last goal, and not in any structure, are *locals*.

The difference between OPAM's classification scheme and that of the PLM is that any variables occurring in the last goal of a clause are always classified as global in OPAM, whereas in the PLM, local variables are permitted in the last goal, which prevents recovery of the local frame before the last call is made (i.e. prevents Last Call Optimisation).[3]

Variables are analysed to reveal first and subsequent usage. Variables whose first occurrence is in a structure or in the body of the clause must be initialised before use, but variables whose first occurrence is in the head, and not in a structure, need not be.

4.5 Representation of Prolog in OPAM Code

In this section is described the scheme by which Prolog source code is represented in OPAM code. Figure 4.6 on page 94 shows how the standard `concatenate/3` predicate is encoded, and figure 4.7 on page 95 depicts how the OPAM code is laid out in memory.

[3]Subsequent to the PLM, Warren made extensive changes to DEC10-Prolog [74, 76], introducing argument registers to facilitate Last Call Optimisation. This new architecture, called Prolog-10 by Tick [69], seems to be an intermediate architecture between the PLM and the WAM, based on structure sharing like the PLM, but using argument registers similar to the WAM.

Term Code		
`local`	`refL`	Local Variables
`global`	`void`	Global & Void Variables
`var`	`varLand`	Global Variables
`atom`	`atomLand`	Atoms
`integer`	`integerLand`	Integers
`structure`	`structureLand`	Structures
Clause Code		
`call`	`lastCall`	Goal Calls
`privateCall`	`privateLastCall`	Private Calls
`neck`	`foot`	Start and end of body
`neckFoot`	`neckCutFoot`	Special Cases
`cut`		The cut symbol
`andCall`	`orCall`	Start of conjunction & disjunction
`catchCall`	`punctuation`	Start of `catch/3` and no-op
`ifThenCall`	`ifThenCommit`	Start & end of *if-then*
`notCall`	`notSucceed`	Start & end of a negation

Table 4.1: OPAM Codes

Every item of Prolog source code is stored as an OPAM instruction in tag-and-data form: a one-word (16-bit) tag followed by zero to two words of data. To avoid the runtime overhead of recognising when a term is being used as a clause, there are two categories of OPAM code: *clause code* for the principal components of clauses and *term code* for everything else. Table 4.1 lists all OPAM tags, both for term code and clause code.

4.5.1 Terms

Simple terms are encoded using a tag-and-data instruction format, as shown in Figure 4.1 on page 85. Instructions comprise a one-word tag followed by between zero and two words of data. Voids need no extra data, local and global variables have one word of data—an offset into the local or global frame respectively—and atoms and integers have two words of data, to store a

longword offset into the name table or a longword signed integer respectively.

A structure is represented by a combination of a structure instruction and a structure sequence. The structure instruction comprises the `structure` tag followed by a one word offset to a *structure sequence*.

A structure sequence is simply a reference to the structure's principal functor followed by each subterm encoded sequentially in memory, and terminated by word denoting its size. The last subterm of a structure is specially tagged with a `land` tag.[4] Thus, for instance, if the last subterm was an atom, its tag would be `atomLand` as distinct from `atom`, `land`. While this significantly increases the number of tags, and thus the number of instructions the OPAM must be capable of executing, considerable space and execution time is saved by this optimisation, particularly with structures such as lists.

4.5.2 Clauses

Clause Code

Given that Prolog clauses are themselves terms, it would be possible to represent clauses in term code. However, it is very advantageous to represent clauses specially. Features of a clause can be detected at compile time which, if encoded suitably, can save time and memory at run time. As with the PLM, a clause is viewed as having head arguments, a neck, possibly a body and a foot, and extra OPAM instructions are used to represent those components of a clause. This extra code is called *clause code*; since clauses contain terms, a clause will be represented by a combination of clause code and term code.

Clauses are compiled separately and converted into a sequence of OPAM instructions and a descriptor record called a *clause descriptor record*. In

[4]This term is borrowed from the name used for the mark left on the cylinder wall of an engine by the piston rings; it signifies the limit of the piston rings' movement. Analogously, it marks the limit of a structure's subterms.

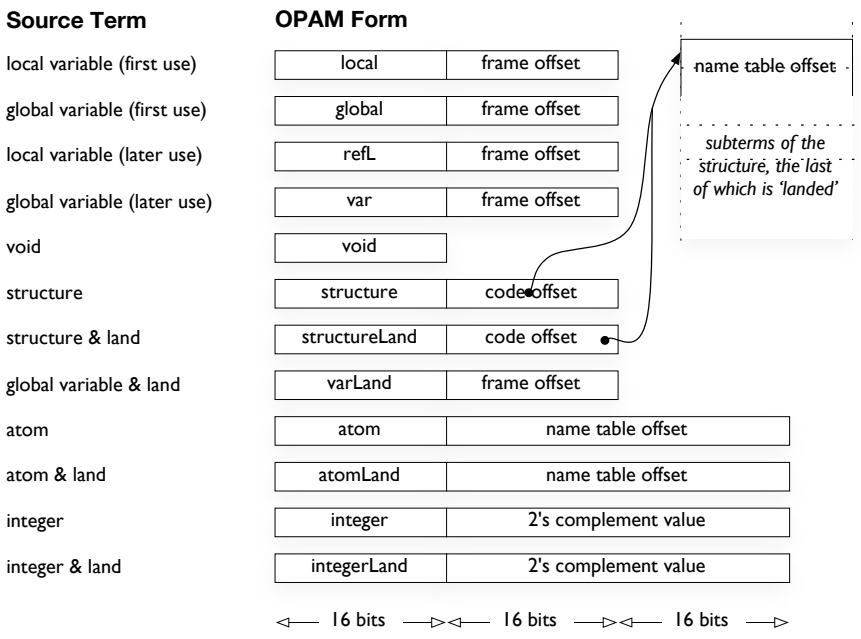

Figure 4.1: The complete set of OPAM Term Code Instructions. Since local variables and voids can not be part of a structure, there is no need for a landed variant of those tags.

Figure 4.2: In the OPAM, a clause is encoded into a descriptor called the *clause descriptor record*, head code, neck, body code, foot and the code size, followed by structure sequences.

this section the encoding scheme is introduced and discussed. A clause is encoded into a head part, a neck, a body part and a foot, preceded by a clause descriptor record and followed by a word denoting the size of the code, as shown in Figure 4.2.

The Clause Head

The head code represents the arguments of the principal functor. Structure sequences are stored at the end of the clause. At the end of the head code, the first clause code instruction we encounter is the `neck` instruction. This represents the neck of the clause, and is responsible for initialising part of the clause instance's environment. The instruction takes three arguments: the size of the global frame, the number of local variables yet to be initialised, less one, and the total number of bytes allocated for local variables.

The Clause Body

A clause body comprises a sequence of predicate calls, cuts, the standard control constructs in Prolog: *negation*, *nested conjunction*, *disjunction*, *if-then* and *if-then-else*, and finally, the predicate `catch/3`, specially encoded, may be present in the body of a clause. Each of these items is encoded uniformly, irrespective of the level of nesting within control constructs. This approach facilitates high performance and simplifies implementation of the inter-related semantics of the cut and the control constructs. The conjunction of terms in the clause body is not explicitly encoded—it is implicit in the sequence of the OPAM encodings.

Public and Private Predicates

Clauses loaded into Open Prolog by the user are *public*—they can be inspected, debugged, modified, supplanted or replaced. However, much of the Open Prolog runtime system is itself written in Prolog, and it is desirable that this *private* code should be hidden and protected from modification or deletion. Private code is almost identical to public code. The difference is that calls made from private clauses to other private clauses bypass the Name Table; that is, they reference their target predicates' first clauses directly, without having to locate them through the Name Table. Thus, changes to Name Table entries can not affect private calls. Built-in predicates written in Prolog are compiled into private code; however, an unmodifiable reference to each such predicate's [first] clause is placed in the Name Table to enable public clauses to reference it.

Predicate Calls

Predicate calls are encoded as shown in Figure 4.3. A call is encoded with a `call` or `lastCall` tag followed by a one-word offset to a sequence that begins with a reference to the predicate in the name table, is followed by the call arguments and is terminated by a `land` and a size word. This is similar to the encoding scheme for a structure, where the `structure` or `structureLand` tag is replaced by a `call` or `lastCall` tag. There are three differences however. First, to assist implementation of the OPAM emulator, the `land` tag is not merged into the tag of the call sequence's last argument, as would be the case with the last subterm of a structure sequence. Second, this encoding scheme is used even if the call is to a predicate with no arguments, which might otherwise be expected to be encoded similarly to an atom. Third, Open Prolog allows for private calls to private predicates—predicates that cannot be listed or modified by the user. Private calls bypass the name table; they

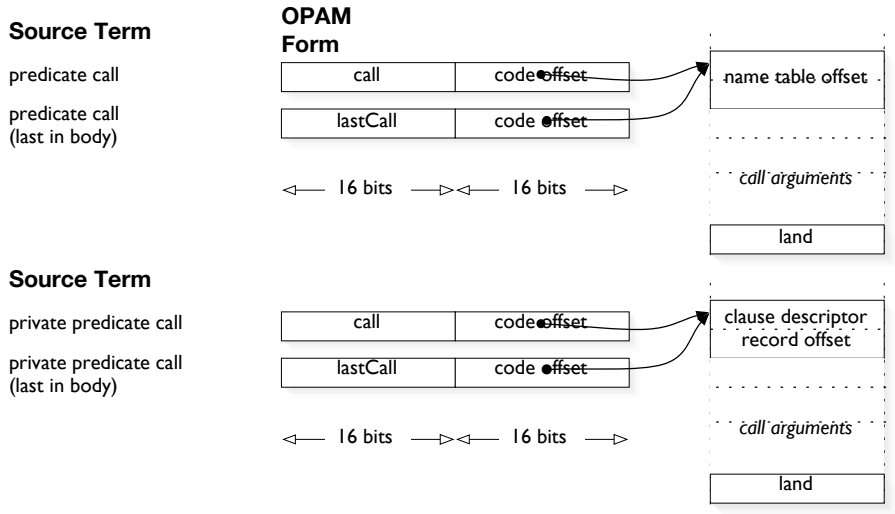

Figure 4.3: Predicate calls are encoded as a `call`/`lastCall` tag followed by a one-word offset to the *call sequence* representing the predicate and its arguments, terminated by a `land` and a size word.

are linked directly to the descriptor of the first clause in the predicate. To facilitate this, the predicate call variants `privateCall` and `privateLastCall` exist. For private calls, the first argument in the call sequence is a pointer to the descriptor of the first clause of the predicate rather than to a name table entry.

Control Constructs and `catch/3`

Common to the control constructs and to `catch/3` is the fact that some of the arguments to these constructs are themselves sequences of calls, cuts and control constructs, the same as might appear in any other part of the body of the clause. Thus, even though lexically the arguments are distinct from the conjunction of clauses that forms the body of the clause, from a procedural point of view they are part of it. The approach adopted therefore was to compile such arguments as if they were part of the clause body. The control constructs are then represented by distinct codes bracketing their compiled

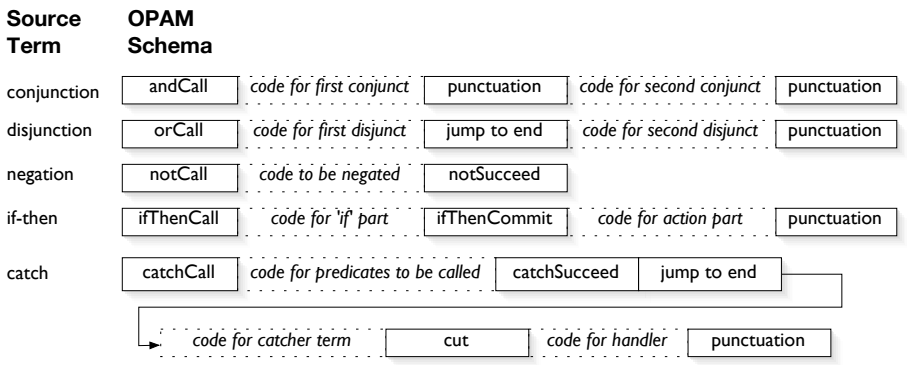

Figure 4.4: Control constructs, like disjunction, and the built-in predicate `catch/3` are encoded according to the schemas depicted.

arguments. Figure 4.4 depicts the schemas used for nested conjunction, disjunction, negation, if-then and the `catch/3` predicate.

For example, consider the clause body:

```
member(X,[a,b]),(jack(X);jill(Y)),write(Y).
```

The disjunction is encoded (in pseudocode) as:

```
        orCall(L1,L2)              %start of disjunction
        call jack(X)
        jump(L2)             %end of first disjunct, start of second
label(L1)
        call jill(Y)
        punctuation          %end of disjunction
label(L2)                    %continuation
        ...
```

Control constructs are encoded as follows:

Nested conjunction. A nested conjunction is preceded by an `andCall`; the encodings for predicates in the conjunction follow and the conjunction is terminated by a `punctuation`.

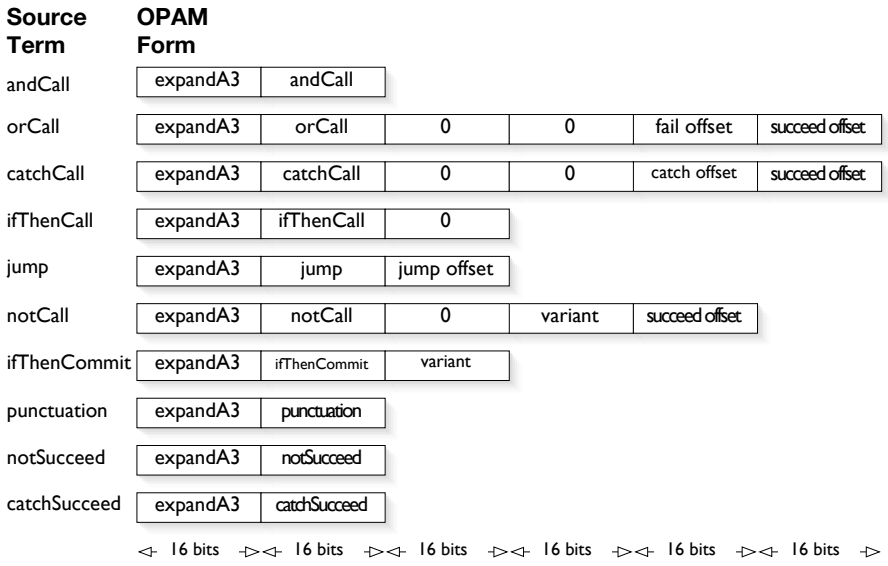

Figure 4.5: Control construct components

Disjunction. A disjunction is begun by an `orCall`; the encoding for the first disjunct follows; next comes a `jump` instruction; this is followed by the encoding for the second disjunct. The disjunction is terminated by a `punctuation` followed by a `label` being the destination of the jump.

Negation. Four negation predicates are recognised in Open Prolog: `\+ /1`, ¬`/1`, `not/1` and `fail_if/1`. They are encoded using separate variants of `notCall`. For example:

```
fail_if(jack(X)).
```

is encoded as:

```
        notCall(3,L1)       %start of negation (variant 3)
        call jack(X)
        notSucceed          %end of negation
label(L1)                   %continuation
```

If-then. This is encoded as shown by the following example:

```
jack(X)->jill(Y).
```

is encoded as:

```
        ifThenCall          %start of if-then
        call jack(X)
        ifThenCommit
        call jill(Y)
        punctuation         %end of if-then
```

The `catch/3` predicate. The three arguments taken by this predicate are, first, a goal to be called, second a term to attempt to unify with if a term is thrown by the goal or its subgoals, and third, a goal to be called if the term thrown unifies with the second term. Thus, the first and third terms are predicates to be treated as first class members of the clause body, and the second argument is a regular term. A catch call is encoded as shown in the following example:

```
catch(special(X),E,write(E))
```

is encoded as:

```
        catchCall(L1,L2)    %start of catch
        call special(X)     %goal called
        catchSucceed
        jump(L2)
L1
        call skip(skip(E)) %E is the term to catch
        cut
        call write(E)       %goal called if term thrown
        punctuation         %end of catch
L2
```

The code descriptions listed above for are control constructs are pseudocode; Figure 4.5 shows the actual encodings used in OPAM.

Clause Descriptor Records

Along with the code representing the clause itself, each clause has a fifty-byte descriptor called a *clause descriptor record* associated with it. This contains information about when the clause was asserted and retracted, a simplified copy of its first argument, fields used during garbage collection, some flags, a pointer to the next clause in the procedure and the size of the clause code in bytes. Table 4.2 details the contents of a clause descriptor record.

4.5.3 Examples

As seems traditional, Figure 4.6 displays a listing of the encoding of the standard `concatenate/3` predicate.[5]

4.6 Data Areas

As a Macintosh application program, Open Prolog is allocated a contiguous range of main memory as its *application zone*. The zone holds the application's stack at the top end and the heap at the bottom. The heap accommodates dynamically allocated memory allocated in relocatable or non-relocatable blocks. When a non-relocatable block is assigned, it is referenced by a pointer to the first byte in the block. A relocatable block is referenced via a 'handle' which is a pointer to a 'master pointer' which points to the present location of the first byte in the block. If a relocatable block is moved by the OS, the master pointer is updated to point to the first byte's new location.

The heap is used extensively by the Mac OS to manage so-called 'resources'. A *resource* is a sequence of bytes of any length, tagged with a

[5]The `concatenate/3` predicate is the same as the `append/3` predicate.

Item	*Function*
Functor Ref	A reference to the name and arity of this clause, zero if the clause is private
Delta	Used during garbage collection. It's the distance the code must be relocated during the 'sweep' phase
Reachable	Used during garbage collection. True if the clause is still alive, false otherwise.
Speed Flags	Used to simplify and speed up clause access during execution
File Number	Not Used
Start Offset	Not Used
Stop Offset	Not Used
Flags	These classify the clause type
Tag	The tag information for the clause's first argument
Data	The data information for the clause's first argument
Assert Time	The database event number when the clause was asserted
Retract Time	The database event number when the clause was retracted
Next	Pointer to the next clause in the procedure, or zero
Code Size	Size of the clause code in bytes

Table 4.2: Fields in a Clause Descriptor Record. The record is fifty bytes long.

```
; code for concatenate([],X,X).
C1:                         ; concatenate(
        -1,                 ; %initialise no globals
        atom([]),           ; []
        local(0),           ; X,
        refL(0),            ; X
        neckFoot(1),        ; ).
        clauseSize(24),

; code for concatenate([H|T],X,[H|R]) :- concatenate(T,X,R).
C2:                         ; concatenate(
        2,                  ; %initialise globals 1, 2 and 3
        structure(L1),      ; <ref> [H|T],
        global(4),          ; X,
        structure(L2),      ; <ref> [H|R],
        neck(5,-1,0),       ; ) :-
        lastCall(L3),       ; <ref> concatenate(T,X,R)
        foot,               ; .
        clauseSize(32),
L1:
        functor(./2),       ; [
        var(1),             ; H|
        varLand(2,12),      ; T] (12 is the size of the structure)
L2:
        functor(./2),       ; [
        var(1),             ; H|
        varLand(3,12),         ; R] (12 is the size of the structure)
L3:
        functor(concatenate/3),  ; concatenate(
        var(2),             ; T,
        var(4),             ; X,
        var(3),             ; R
        land(18)            ; )  (18 is the size of the structure)
```

Figure 4.6: OPAM code for the concatenate/3 predicate. The entry points are labeled C1 and C2.

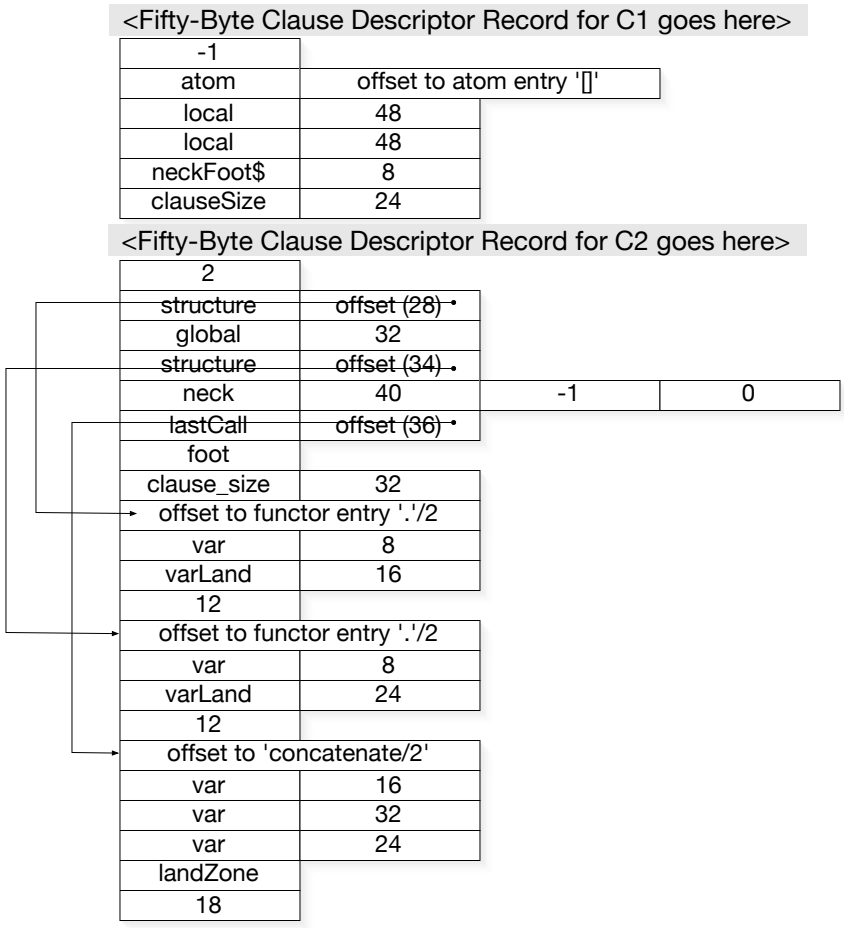

Figure 4.7: Layout of the encoding of the `concatenate/3` clauses. Each short box represents a two-byte item, the longer boxes represent four-byte items. The lines highlighted in grey indicate that each clause's OPAM code is preceded by a fifty-byte Clause Descriptor Record (described on page 92).

four-byte code (often four mnemonic ASCII characters), an index number (a signed 16 bit integer) and one byte of attribute bits used by the Mac OS.

Resources are managed by the Mac OS on behalf of the application. Every file in the Macintosh file system has the potential to have two sections or 'forks', a *data fork* containing an unmanaged stream of bytes, and a *resource fork* which stores only resources. Resources can be loaded into the heap, relocated, modified, written out, and so on. A set of APIs called the Resource Manager is available to facilitate the use of resources and an Apple Computer tool called ResEdit [84] can be used as a graphical inspector and editor. By convention, the executable code of a program is segmented and stored in `CODE` resources, i.e. resources with the four-byte code comprising the ASCII codes of the letters `C O D` and `E`. Moreover, many standard items of data are stored as resources, such as dialog box specifications, pictures, etc. (External predicates are also stored as resources, and are discussed in Chapter 6.2, starting on page 148.) Resources can be preloaded at startup or loaded on demand; they can be locked in place and made immovable; they can be marked 'purgable' so that they can be deallocated by the Mac OS under low memory conditions. `CODE` resources are typically locked and non purgable.

Considerable opportunity therefore exists for serious and sustained memory fragmentation on the heap, and Open Prolog sidesteps this by allocating a large non-relocatable block of memory in the heap as early as possible after startup, as depicted in Figure 4.8. This block, 'OPAM memory', is managed internally by the runtime system, and can change size during memory management to ensure enough space is left in the heap for the rest of the application's needs. Within the OPAM block, five data areas are maintained: the Name Space, the Code Space, the Global Stack the Local Stack and the Trail.

The Name Space contains information about all the functors and atoms known to the program. It comprises two main components—a hash table

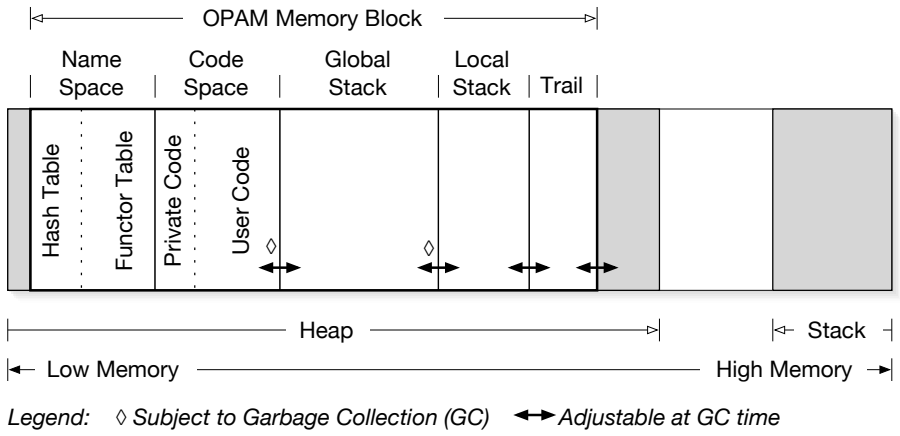

Figure 4.8: The Open Prolog Memory Map. The diagram shows the layout of the complete application zone allocated to Open Prolog by the Mac OS. The OPAM Memory Block is allocated in the heap and partitioned as shown. The heap is used in the normal way to hold relocatable and non-relocatable blocks of memory, including application code and code for built-in and 'external' predicates.

and a functor table. The hash table is conventional, and is used to locate a functor's entry by name and arity in the functor table.

The functor table contains the following information about each individual name: a pointer to first clause (if any), number of clauses, a flags byte containing two flags (`isSpyPoint` is used by the debugger and `isPredicate` indicates if this functor is a predicate), arity, spelling, operator definitions (if any), and most general skeleton, as shown in Figure 4.9. The most general skeleton is used to provide a structure for a term constructed dynamically using the 'univ'[6] and `functor/3` built-in predicates. The length of the most general structure is $8 + arity * 4$ bytes. The Name Space is neither resizable nor garbage-collected.

The Code Space contains the OPAM code for every clause. All private code is loaded at initialisation, and placed at the start of the Code Space.

[6]The 'univ' predicate is used to construct terms from lists and vice-versa. It has the name `=..` and an arity of 2.

Figure 4.9: Layout of an entry in the Functor Table in the Name Space. The Operator Definitions section is present for unary and binary functors.

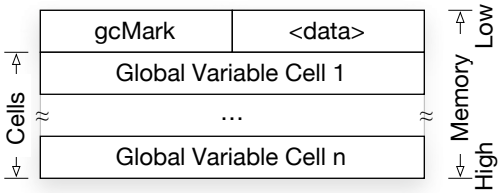

Figure 4.10: The layout of a global frame.

Private code can not be retracted, and so does not need to be garbage collected. User code—the code corresponding to the user's program and clauses asserted during execution—is loaded into the Code Space above the private code. Since user code can be retracted, this part of Code Space is subject to garbage collection and may be resized.

Next to the Code Space is the Global Stack. The Global Stack is used to hold the global frame of each clause instance. A global frame, see Figure 4.10, starts with a special `gcMark` cell (used during garbage collection) and is followed by cells for each global variable. Each cell must be capable of holding a molecule, and is therefore 8 bytes in size. To preserve 64 bit alignments, the `gcMark` cell is also 8 bytes wide. The Global Stack is subject to resizing, garbage collection and remapping, that is, it can be moved up or down within the OPAM Memory Block.

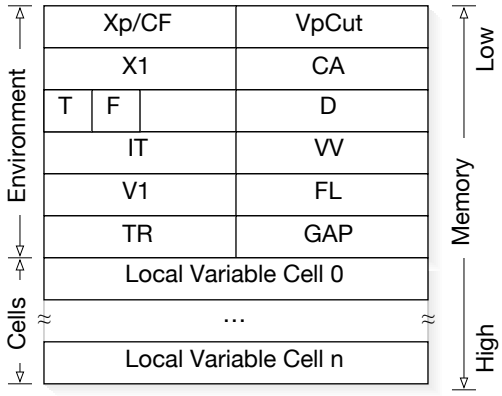

Figure 4.11: The layout of a local frame.

Above the Global Stack is the Local Stack. The Local Stack is used to hold the local frame of each clause instance. A local frame, as shown in Figure 4.11, contains the clause instance's environment followed by cells for each local variable. The `X`, `X1` and `CA` (for Continuation Address) fields are used to restore the `X`, `X1` and `BPC` registers to the values appropriate to the goal within the clause instance at which program execution is continuing. The `VpC` field contains the value to which `Vp` should be set if a cut is executed in the body of this clause instance.

The `T` field contains the one-byte tag of the first goal argument, and the `D` field contains the corresponding data. The fields are valid only if the frame is a choice point and are used upon backtracking to help choose an alternative. The encoding of the first goal argument is slightly different to source term encoding, and is given in Figure 4.12.

The `IT` (Instantiation Time) field records the database event clock value when the clause instance was invoked. This is used during garbage collection. The `VV` field contains a pointer to the next oldest choice point to this one, and the `FL` field contains the address of the next alternative clause at this backtrack point, or zero otherwise. The `V1` and `TR` fields contain values for

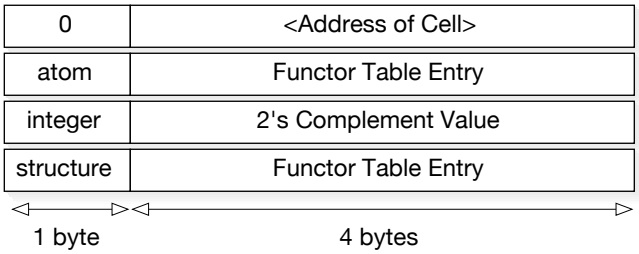

Figure 4.12: Encoding of the first argument of the calling goal. This is stored in the local frame if it is a choice point. Notice that, in the case of a structure, just the principal functor is encoded .

Bit(s)	*Significance*
7	This is the notFirstUseBit, used by certain 68000 built-in predicates
6	Set if the frame is 'transparent' to cuts
5–0	If the frame is special, these bits encode its kind, as follows:
Code	*Kind of Frame*
000001	OrFrame
000010	NotFrame
000011	IfThenFrame
000100	CatchFrame

Table 4.3: Flag Byte of a Local Stack Frame. In a normal frame, all bits in the Flag Byte are zero. The use of the notFirstUseBit is explained in section 6.4 and the use of the other fields is explained in Section 7 on page 159.

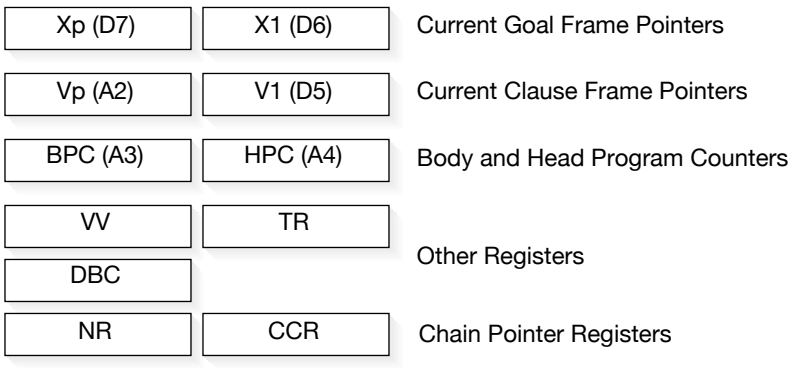

Figure 4.13: The OPAM Register Set. Each register is 32 bits long. The topmost six registers are implemented with the 68000 registers shown in brackets. `Vp` and `V1` point to the top of the local and global stacks respectively, where new clause instance frames are constructed.

register `V1` and `TR`, to be loaded into them on backtracking, and similarly, the `GAP` (Goal Argument Pointer) contains the value to be loaded into the `BPC` register on backtracking to this choice point. The `F` field is a one-byte flags field, indicating the category and other properties of the frame, as shown in Table 4.3.

4.7 Registers

The OPAM has eleven registers, as shown in Figure 4.13. Each register is 32 bits long. `Xp` and `X1` point to the global and local frames of the clause instance from the body of which the current goal is being called. Hence, `Xp` and `X1` specify the current goal's environment.

`Vp` points to the top of the local stack, and `V1` points to the top of the global stack. While a goal is being unified to a clause head, a new clause instance is being created, and its frames are being built on the top of the stacks. Thus, `Vp` and `V1` also point to the local and global frames of the clause instance under construction, i.e. the current clause instance.

`BPC` and `HPC` are program counters to the current goal and the current

clause instance respectively. The BPC is always pointing to part of the body of the current goal's clause instance, hence BPC, the *Body Program Counter*. Similarly, HPC is always pointing to part of the head of the current clause instance, hence HPC, the *Head Program Counter*.

VV points to the first backtrack frame or choice point. TR points to the top of the Trail, used to record variable cells that must be explicitly reset upon backtracking. DBC is a database event clock. The clock is incremented every time a clause (sometimes, a group of clauses) is asserted or retracted, and is used to implement the 'logical' assert and retract semantics of Lindholm & O'Keefe [49].

NR and CCR point to the tops of two 'chains' of local stack frames that need special treatment; NR points to the start of a chain of *non removable frames*, also called *NR frames* and CCR points to the start of a series of local stack frames called the *Catch Chain* Each chain is a one-way linked list of frames where each frame contains a link to the next oldest frame in the chain. NR frames are used in the implementation of control constructs such as negation, if-then and if-then-else, and also in the implementation of catch and throw. Each frame in the catch chain points to the local stack frame of a catch/3 predicate and to the next oldest frame in the catch chain. When an exception is thrown, the catch chain, accessed via the CCR, is used to find the newest, i.e. most local, exception handler capable of handling the exception.

The host processor, a Motorola 68000, has sixteen general purpose registers: eight *data registers*, D0–D7, and eight *address registers*, A0–A7. Of these, A7 is used by the processor itself as a stack pointer and A5 is used by the Mac OS, leaving a total of fourteen registers. Of these, seven (A0–A2 and D0–D3) are 'volatile' across calls to the Mac OS, i.e. their contents are not preserved (see section 4.10). This leaves just three non-volatile address registers and four non-volatile data registers of which six are used to implement OPAM

processor registers.[7] For example, `Vp` is implemented in `A2` (see Figure 4.13 on page 101).

Naturally, it would have been preferable to implement all OPAM registers in host processor registers, but the 68000's register set just isn't big enough, so an attempt was made to select the most heavily used OPAM registers to be implemented in host processor registers.

4.8 Operation

As mentioned in the overview (page 81), the OPAM has two program counters, the `BPC` (Body PC) and the `HPC` (Head PC). These are used in parallel while the OPAM is attempting to unify the arguments of the goal with the arguments of the head of a clause instance. To unify two corresponding arguments (one in the goal and one in the clause head), the tag of the goal argument is fetched via the `BPC` and the tag of the head argument is fetched via the `HPC`. The two tags combine to form an OPAM instruction which is then executed. Another way of putting this is that while in this 'dual PC mode', the OPAM fetches *instruction fragments* via the `BPC` and the `HPC`; the fragments are then combined and executed. If the fragments represent terms that are unifiable, execution is successful. For example, if the goal term is `atom(a)` and the head term is `atom(a)`, then the instruction is `atomAtom(a,a)` and succeeds. Likewise, if the goal term is `integer(21)` and the head term is `local(0)`, then the instruction is `integerLocal(21,0)`, and the instruction succeeds by assigning the value 21 to local variable 0 (the tag `local` means this is the first occurrence of the local variable, so it doesn't need to be dereferenced; it simply needs to be assigned). If the terms are not

[7] When the implementation of Open Prolog was started, it was thought that this non-volatility would be very important for speed, as it would allow access to, say, built-in predicate code without having to save and restore the OPAM processor's register set. In hindsight, it may not have been all that important at all.

unifiable, then execution fails. For example, if the goal term is `integer(21)` and the head term is `structure(concatenate/3)`, the resulting instruction `integerStructure(21,concatenate/3)` fails immediately.

When the OPAM executes structures, it recursively executes their subterms, having ensured the principal functors are identical. The end of a sequence of subterms, indicated by a tag incorporating a `land`, unwinds the recursion.

Along with the dual PC mode of operation, the OPAM has two single-PC modes of operation: executing instructions referenced via the `BPC`—'BPC mode'—and executing instructions referenced via the `HPC`—'HPC mode'. Most of the time, the OPAM alternates between dual PC mode and BPC mode, with HPC mode being used very seldom; call instructions switch from BPC mode to dual PC mode and neck instructions switch back to BPC mode.

Consider the call `concatenate([a],[b],X)`, encoded as in Figure 4.14. The sequence of instructions executed as a consequence of this call is as shown in Figure 4.15 on page 105.

4.8.1 Clause Indexing and the Call Instructions

One of the design aims for Open Prolog was that programs should be easy to modify. In a Prolog context, that means that it should be possible to assert and retract clauses, just as in a more conventional interpreter. If one wishes to use some form of clause indexing (e.g. to detect determinacy), then a problem immediately arises: one can not be certain that just one clause in a procedure is viable, (and thus that the call is determinate), if another clause could be added later. Even worse, a clause that formed part of procedure could be removed, potentially leaving a choice point behind that referred to it.

A coherent and practical solution to this problem was developed by Lind-

```
        call(L1),           ; call <ref> concatenate([a],[b],X).
        ...
        ...
L1:
        functor(concatenate/3),  ; concatenate(
        structure(L2),      ; <ref> [a]
        structure(L3),      ; <ref> [b]
        var(1),             ; X
        land(18),           ; )
L2:
        functor(./2),       ; [
        atom(a),            ; a |
        atomLand([],16),    ; [] ]
L3:
        functor(./2),       ; [
        atom(b),            ; b |
        atomLand([],16)     ; [] ]
```

Figure 4.14: Sample call to `concatenate/3`. The call is `concatenate([a],[b],X)`.

```
;;      Instruction                 ; PC Mode    ; Level
        call(concatenate/3)             ; BPC->Dual ; 0
        structureStructure(./2,./2) ; Dual       ; 0->1
        atomVar(a,1)                ; Dual       ; 1
        atomLandVarLand([],2)       ; Dual       ; 1->0
        structureGlobal(./2,4)      ; Dual       ; 0
        varStructure(1,./2)         ; Dual       ; 0
        landNeck(5,-1,0)            ; Dual->BPC  ; 0
        lastCall(concatenate/3)         ; BPC        ; 0
        varAtom(2,[])               ; Dual       ; 0
        varLocal(4,0)               ; Dual       ; 0
        varRefL(3,0)                ; Dual       ; 0
        landNeckFoot(1)             ; Dual->BPC  ; 0
        ...                         ; BPC        ; 0
```

Figure 4.15: This is the sequence of instructions executed by the call `concatenate([a],[b],X)` in Figure 4.14.

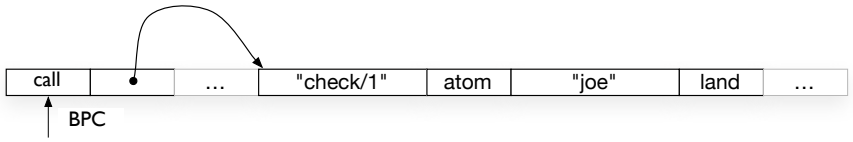

Figure 4.16: The BPC points to the next instruction to be fetched, a `call` instruction followed by its operand and by subsequent instructions. The call's one-word operand is an offset to the call sequence—the predicate's functor entry followed by its arguments and terminated by a `land`.

holm and O'Keefe in [49] using a database clock[8] and time stamps to indicate when a clause is asserted and retracted and which clauses are or were in existence when a call or retry is made.

In this section, the operation of the call instructions is explained. These instructions are at the heart of the operation of the OPAM, and are important in the implementation of last call optimisation and the memory management of the code space.

A high proportion of OPAM execution time is spent executing procedure calls and a number of call instructions exist to reduce that time by taking advantage of special cases. The call instructions are `call`, `lastCall`, `privateCall` and `privateLastCall`. Additionally, much of the functionality of the call operation is delegated to *call action procedures*, as will be detailed in the following sections.

We begin examination of the operation of the OPAM just when a new goal is about to be called. Figure 4.16 displays a schematic of a call to the predicate `check/1`. The call, `check(joe)`, is pointed to by the `BPC`.

Operation of the Call Instructions

The call instructions operate as follows:

[8]It seems traditional to refer to the clause store as a 'database', irrespective of how it is actually implemented.

- The continuation is identified. The continuation is the next goal to be executed once the current goal has been completed. Normally, the continuation is simply the goal following the current one in the body of the parent clause. However, if the current goal is the last goal in the body of the parent clause (i.e. if the instruction is `lastCall` or `privateLastCall`), then the continuation is the parent's own continuation. This is one part of the implementation of Last Call Optimisation.

- If the current goal is the last goal, and if program execution is determinate, then the memory allocated to the local frame of the current goal's clause instance (the 'current local frame' in the sequel) is recovered and may be reallocated. This is because the following conditions hold:

  - The frame of the current goal's clause instance is effectively at the top of the local stack. Since program execution is determinate, the most recent choice point must be older than the current local frame. Likewise, the continuation frame must either be the current local frame's direct parent or an older ancestor, so it is deeper within the stack. Hence, frames newer than the current local frame can be discarded and their space recovered.

  - The current local frame itself is recoverable, because nothing remaining in it is needed:

    * None of the variables in the last goal are local, so no local variable cells are used.

    * None of the choice point information stored in the local frame will be used because the program is determinate at this point.

    * The only piece of important information in the frame is the continuation address, which will have been processed in the first step above.

Thus, nothing in the current local frame is needed, and the memory allocated to the frame is recovered. This is the second part of the implementation of Last Call Optimisation.

- The procedure to be called is located. Recall that the clause descriptor records of all clauses in the same procedure (i.e. clauses with the same principal functor) are linked together. If the procedure is public, the clause descriptor record of the first clause in the chain is obtainable via the Name Table entry of the procedure's principal functor. Accordingly, the `call` and `lastCall` instructions use the functor reference at the start of the call sequence to locate the functor's entry in the name table, and from there the location of the clause descriptor record of the first clause in the procedure is retrieved. (If no clause exists and the `isPredicate` flag is set, the call fails; otherwise, an unknown predicate exception is thrown.)

  The `privateCall` and `privateLastCall` instructions operate slightly differently. When private calls are linked to private procedures at system startup, the functor at the start of each private call sequence is replaced by a reference to the clause descriptor record of the first clause in the private procedure.[9] Private calls therefore bypass the name table completely, and when a `privateCall` or `privateLastCall` is executed, the location of the clause descriptor record of the first clause in the procedure is retrieved from the start of the private call sequence.

- At this point, the call instruction has:

  - Located the clause descriptor record of the first clause in the procedure being called,

[9] This is safe because private procedures are never retracted or moved by garbage collection.

– Located the continuation address and the continuation frame to be used after this procedure has exited,

– Recovered local stack space in the case of a determinate last call.

The call instruction now locates the *call action procedure* in the first clause's clause descriptor record and transfers execution to it.

This completes the operation of the call instruction, but a good deal of further processing may yet be performed by the call action procedure before execution begins on a clause.

Operation of the Call Action Procedure

The call action procedure is a parameter within the clause descriptor record of a clause, so it can be tailored to the clause's requirements; in the current implementation, though, the call action procedure of the first clause is used for all the clauses in the procedure.

The standard call action procedure, which is the most general action procedure in use, searches the linked list of clause descriptor records for up to two 'live' clauses whose first argument might unify with that of the goal.

The check for liveness is part of the implementation of the 'logical' database assert/retract semantics of Lindholm and O'Keefe.[49] A clause is 'live' if the current value of the `DBC` register is bracketed by the clause's assertion date and retraction date—in other words, if the clause was asserted before the current time and has not yet been retracted, it is live.

The standard action procedure calculates the goal's first argument tag and data information if necessary, and checks it against each candidate. If a clause is 'live', as described above, and if its First Argument Tag and First Argument Data (part of its clause descriptor record) match the goal's first argument tag and data, or if either term is a variable, then the clause is selected. The standard action procedure thus selects a live clause whose first

argument might match the goal's first argument. In addition, a candidate alternate clause is identified, if it exists, so it is known if a choice point needs to be constructed.

At this point, the Continuation Address (effectively the return address) is stored in the `CP` field of the local frame. If the frame is to be a choice point, then the additional information necessary to restore the environment on backtracking is placed in the local frame:

- The goal's first argument tag and data, already calculated, are stored in the `T` and `D` fields respectively;
- The current value of the `DBC` register is stored in the `IT` field, to denote the database instant at which the goal was called. This is part of the infrastructure needed to implement 'logical' database semantics;
- The address of the alternative clause is stored in the `FL` field;
- The address of the sequence of goal arguments is stored in the `GAP` field;

The standard call action procedure initialises the number of global variables specified as the first part of the clause, starting from the bottom of the global frame, before finally changing OPAM to dual PC mode.

Specialised Call Action Procedures

The full rigours of the standard action procedure are unnecessary in a number of important situations. For instance, if there is exactly one clause in a procedure, it is not really useful to calculate or check for first argument matching. This situation is accommodated within the standard action procedure.

If the clauses are private (and so cannot be retracted), or if it is otherwise known that no clauses have been asserted or retracted, then the expensive check for liveness can be omitted. This improvement is accommodated in

a call action procedure called the 'faster action procedure'. After garbage collection, the standard call action procedures of clauses in a procedure that contains no remaining retracted clauses are replaced by the faster action procedure, and whenever a new clause is added to a procedure, the standard call action procedure is attached to each clause.

Another important situation is where a call is being made to a built-in predicate. Built-in predicates can't be retracted and in many cases there will only be one clause. Furthermore, the predicate code itself may be able to check the goal's first argument faster than the standard action procedure. Thus, none of the functionality of the call action procedures is useful, and represents useless overhead. In these cases, therefore, the call action procedure is omitted and replaced by the built-in predicate code itself. That is, once the call instruction has identified the continuation address and frame and has identified the predicate to be executed, the predicate's code is immediately called. This fast dispatch mechanism is a considerable improvement for many built-in predicates, where the amount of processing to be done is very small and where the overhead of a normal call action procedure would be disproportionately high.

4.9 Extended Example

This example is identical to the standard naive reverse benchmark except that the list has just two elements. It will be referred to later during the discussion of profiling. The example consists of the clauses listed in Figure 4.9.

These are compiled into three procedures, `nreverse/0`, `nreverse/2` and `concatenate/3`, show in figures 4.18, 4.19 and 4.6 respectively.

Executing the goal `nreverse` gives rise to a sequence of operations shown in Figure 4.20. It shows the standard [Prolog] trace of the execution of the goal `nreverse` annotated with the trace of the execution of the corresponding

```
nreverse :- nreverse([1,2],_).

nreverse([X|L0],L) :- nreverse(L0,L1), concatenate(L1,[X],L).
nreverse([],[]).

concatenate([X|L1],L2,[X|L3]) :- concatenate(L1,L2,L3).
concatenate([],L,L).
```

Figure 4.17: Extended Example of OPAM Encoding. These clauses are encoded in OPAM code as shown in Figures 4.18, 4.19 and 4.6.

```
;code for nreverse :- nreverse([1,2],_).
C3:                             ; nreverse
        -1                      ; %initialise no globals
        neck(1,-1,0)            : :-
        lastCall(L4)            ; <ref> nreverse([1,2],_).
        foot                    ;
        clauseSize(20)          ;

L4:
        functor(nreverse/2)     ; nreverse(
        structure(L5)           ; [1,2]
        void                    ;
        land(12)

L5:     functor(./2)            ; [
        int(1)                  ; 1|
        structureLand(L6,14)    ; <ref> [2]] (14 is size of structure)

L6:
        functor(./2)            ; [
        int(2)                  ; 2 |
        atomLand([],16)         ; []]
```

Figure 4.18: OPAM code for the `nreverse/0` predicate.

```
;code for nreverse([X|L0],L) :- nreverse(L0,L1), concatenate(L1,[X],L).
C4:                             ; nreverse/2(
        2                       ; initialise 3 globals
        structure(L7)           ; <ref> [X|L0]
        global(4)               ; L
        neck(5,-1,0)            ; ) :-
        call(L8)                ; <ref> nreverse(L0,L1)
        lastCall(L9)            ; <ref> concatenate(L1,[X],L)
        foot
        clause_size(32)

L7:
        functor(./2)            ; [
        var(1)                  ; X|
        varLand(2,12)           ; L0]

L8:
        functor(nreverse/2)     ; nreverse(
        var(2)                  ; L0,
        var(3)                  ; L1
        land(14)                ; )

L9:
        functor(concatenate/3)  ; concatenate(
        var(3)                  ; L1,
        structure(L10)          ; <ref> [X]
        var(4)                  ; L
        land(18)                ; )

L10:
        functor(./2)            ; [
        var(1)                  ; X
        atomLand([],14)         ; | [] ]

;code for nreverse([],[]).

C5:                             ; nreverse(
        -1                      ; initialise no globals
        atom([])                ; []
        atom([])                ; []
        neckFoot(1)             ; ).
        clause_size(22)
```


Figure 4.19: OPAM code for the nreverse/2 predicate.

OPAM instructions.

4.10 Runtime Environment

As a Macintosh application, Open Prolog conforms to a number of conventions and requirements:

- The code is Motorola 68000 code, with a register usage convention:
  - `A7` is always the Stack Pointer
  - `A5` is always used to reference global variables,
  - `A6` is often used as a stack frame pointer
  - Registers `A0`–`A2` and `D0`–`D3` are volatile across an API call or across a call to code compiled from a high level language.

  Thus, of the 16 processor registers available, two are used by the Mac OS and a further seven are volatile across API and high-level routine calls. As many of Open Prolog's OPAM registers as possible are mapped to the non-volatile 68000 registers, and the OPAM register `Vp` is mapped to `A2`.

- The application is given a fixed amount of memory—its 'application zone'—to accommodate the stack and heap. It is possible to borrow extra memory from time to time, but the application must basically operate within the fixed allocation. A further issue is that the allocation and deallocation of memory is time-consuming, and may well fail due to memory fragmentation, and is best avoided. To minimise these difficulties, Open Prolog allocates a proportion of the zone to the stack at startup, allocates most of the heap to the OPAM Memory Block, and leaves the rest for the normal use of the Mac OS.

```
Prolog Execution Trace                OPAM Instruction Sequence
1 | 1 call nreverse                   call%(nreverse/0)
                                       neck%
2 | 2 call nreverse([1,2],A)          lastCall%(nreverse/2)
                                       structureStructure(./2,./2)
                                       integerVar(1,1)
                                       structureLandVarLand(./2,2)
                                       voidGlobal(4)
                                       neck%
3 | 3 call nreverse([2],B)            call%(nreverse/2)
                                       varStructure(1,./2)
                                       integerVar(2,1)
                                       atomLandVarLand([],2)
                                       globalGlobal(2,4)
                                       neck%
4 | 4 call nreverse([],C)             call%(nreverse/2)
                                       varAtom(1,[])
                                       globalAtom(2,[])
4 | 4 exit nreverse([],[])            neckFoot%
5 | 4 call concatenate([],[2],B)      lastCall%(concatenate/3)
                                       varAtom(3,[])
                                       structureLocal(./2,0)
                                       varRefL(4,0)
5 | 4 exit concatenate([],[2],[2]) neckFoot%
3 | 3 exit nreverse([2],[2])
6 | 3 call concatenate([2],[1],A)  lastCall(concatenate/3)
                                       varStructure(3,./2)
                                       varVar(1,1)
                                       atomLandVarLand([],2)
                                       structureGlobal(./2,4)
                                       varStructure(4,./2)
                                       neck%
7 | 4 call concatenate([],[1],D)      lastCall%(concatenate/3)
                                       varAtom(2,[])
                                       varLocal(4,0)
                                       varRefL(3,0)
7 | 4 exit concatenate([],[1],[1]) neckFoot%
6 | 3 exit concatenate([2],[1],[2,1])
2 | 2 exit nreverse([1,2],[2,1])
1 | 1 exit nreverse
```

Figure 4.20: Prolog trace and OPAM execution trace of OPAM code of `nreverse/0`.

- The application has a user interface that is responsive at all times, but the Mac OS doesn't support multiprocessing. This is addressed in Open Prolog using a section of code called the 'juggle code' described in section 4.10.2 below.

4.10.1 OPAM Emulation Code

The instructions and core facilities (e.g. the garbage collector) of the OPAM are implemented in Motorola M68000 Assembly Language, as is much of the user interface code. Most of the rest is written in Pascal, with a small amount written in C. The development system was MPW, the Macintosh Programmers Workshop [82].

At the heart of the machine is the dispatch mechanism, the code used to dispatch the next OPAM instruction. The machine operates as an indirect threaded code [29] interpreter in one of three modes: dispatching an instruction whose components are referenced by two program counters, or dispatching an instruction from one or the other program counter. The program counters are implemented in the 68000 by `A4` for the Head Program Counter (`HPC`) and `A3` for the Body Program Counter (`BPC`). To dispatch an instruction in Dual-PC mode, the body fragment is fetched and used to select the dispatch table appropriate for that fragment. For example, if the fragment is `atom` then the 'Atom' dispatch table is chosen. The head fragment is then fetched and used to select an entry in the chosen dispatch table. The entry chosen is the address of the first instruction of the emulator code for that instruction. Thus, continuing the example, if the head fragment is `local`, the address of the emulation code for the instruction `atomLocal` is selected.

Every OPAM instruction has a section of emulator code, and each section consists of the code that actually emulates the OPAM instruction followed

by the dispatch code for fetching the next instruction. For example, here is the code for the OPAM instruction `atomGlobal`, with the last four lines comprising the dispatch code:

```
;atomGlobal
    move.w  (a4)+,d0
    ext.l   d0
    move.l  d0,a0
    add.l   V1,a0
    moveq   #atom,d0
    move.l  d0,(a0)+
    move.l  (a3)+,(a0)+

    move.w  (a3)+,d0
    move.w  primaryDispatchTableBottom-zeroDataOffset(a5,d0.w),d0
    move.l  -4(a5,d0.w),a0
    jmp (a0)
```

Related Work

Krall and Neumerkel independently developed a similar arrangement for a structure copying implementation called the Vienna Abstract Machine (the VAM$_{2P}$ [47]).[10] Where the OPAM has a `HPC` and a `BPC`, the VAM$_{2P}$ has the `headptr` and the `goalptr`. In the VAM, terms in the head and goal are encoded differently so that simply adding the head term and goal term gives a unique result that can be used to reference the routine needed. On the other hand, part of the motivation of dual PCs in Open Prolog was to *preserve* the uniformity of the encoding of terms, be they in the head or in the body of a clause.

In assembly language, the VAM$_{2P}$ is implemented using direct threaded code [6], and in C, a `switch` statement is used. Garbage collection is performed, but there does not appear to be support for dynamic clauses or

[10]The subscript $_{2P}$ indicates a two-PC implementation, and a single-PC variant, the VAM$_{1P}$, was also described.

associated garbage collection.

Xining Li describes an emulator based on a representational technique called *Program Sharing* [48] in which an emulator with two PCs is used for unification and one PC for executing 'control' instructions.

Development Environment

The MPW development offers macro assembler, and so it was possible to devise macros for many of the repetitive sections of the emulator code; for instance, the code for `atomGlobal` above is almost the same as the code for `atomLocal`, `integerGlobal` and `integerLocal`, and a macro was used to define the common code once. In all, the 'engine' part of Open Prolog, comprising the emulation code for the OPAM instructions, is a little less than 4,000 lines of macro assembler. The inner core of the implementation, comprising the emulator, the garbage collector and a subset of the faster built-in predicates, is written in less than 10,000 lines of macro assembler.

4.10.2 User Interface

As a Macintosh application, Open Prolog must have a user interface that is responsive at all times. A convenient way to organise this would be to have two separate threads (or processes), one running a user interface and the other running the conventional Prolog listener. Data would be transferred between the threads during interaction with the user. The problem is that the Mac OS does not support threads, processes or even pre-emptive multiprocessing, so the application had to be written to explicitly allow the user interface to share the processor with the Prolog listener at frequent intervals. This is done in a slightly unconventional way, to minimise the overhead required. The event processing code is written as a subroutine called `userInterface`. While the Prolog listener is waiting for input from the user, `userInterface` is repeatedly called, implementing the conventional event

loop. However, if the Prolog listener is executing a program, then a timed interrupt every one hundred milliseconds (fifty milliseconds if Open Prolog is running in the background) 'diverts' the OPAM emulator's dispatch code. The diversion redirects the dispatch of `call%`, `lastCall%`, `privateCall%` and `privateLastCall%` OPAM instructions to a section of code called the 'juggle code'. The juggle code calls the `userInterface` subroutine to service the user interface. On exit, the juggle code resets the emulator's dispatch mechanism and executes the OPAM instruction originally dispatched.

Effectively, therefore, if the Prolog listener is idle, the user interface event loop runs in the normal way. On the other hand, if it's executing a Prolog program, user interface events are processed every hundred milliseconds.

The user interface code deals with everything that doesn't need the Prolog listener's involvement, including text editing, menu handling, co-operative multitasking, etc.

4.10.3 Program Startup

At startup, Open Prolog firstly attempts to maximise the amount of contiguous free space available in the Application Zone (the memory space assigned to it by the Mac OS). Next, it reserves the OPAM Memory Block (see section 4.6) and initialises it to contain the Name Space, Code Space, Global Stack, Local Stack and Trail areas. Next, resources containing OPAM and 68000 code—comprising support code, built-in predicates and 'external' predicates—are located and loaded, linked and initialised, and execution is transferred to the traditional Prolog listener communicating with the user through a text-oriented window-based user interface. Each time through the user interface event loop, pending 'interrupts' from external code will be processed, and interrupt handlers will be launched if appropriate.

4.11 Compilation

Clauses are compiled individually, each clause being converted into a sequence of OPAM codes and a clause descriptor record. Procedures are formed by linking the clause descriptor records of clauses having the same principal functors together. Names—atoms and functors—are stored in a name table. Included with each name is a reference to the clause descriptor record of the first clause bearing its name. The principal functor is not needed in the clause code itself, and is omitted. Private clauses can exist without any references in the name table; they are accessed directly by their callers using `privateCall` and `privateLastCall` clause code. Linking of private calls to private clauses is done at startup time.

4.12 Memory Management

The size of the Name Space is fixed at startup; it can not be adjusted after that time, so it is possible to overflow the Name Space. All the other spaces—Code Space, Global Stack, Local Stack and Trail Stack—are actively managed at runtime. If any one of them runs low on spare capacity, memory management is performed.

- If sufficient spare capacity is available elsewhere, the space allocations are adjusted, moving the stacks around as necessary.
- If there is not sufficient free space, garbage collection is performed on the Global Stack and on the Code Space, after which stack allocations will be adjusted. If insufficient space is available after a garbage collection, the system aborts.

  Memory management is discussed in detail in Chapter 8, on page 176.

4.13 Built-In Predicates

Open Prolog has a fairly extensive set of built-in predicates, somewhat compatible with ISO Prolog. Many of these predicates are implemented at least partly in Prolog. The rest are implemented in host (i.e. Motorola 68000) code, either written in assembly language, in Pascal or in C. An external compiler is used to compile built-in predicates written in Prolog into OPAM code. These are stored as Mac OS `PRLC` resources for access at startup time. In addition, many procedural built-in predicates are assembled or compiled as part of the complete system, and are stored as `PRLC` and `CODE` resources. A special category of host-coded built-in predicates, called 'External Predicates' are compiled completely separately to the system. These routines too are stored as `PRLX` resources.

The implementation of built-in predicates is discussed in Chapter 6, on page 148.

4.13.1 Private Built-In Predicates

As mentioned in Section 4.5.2 on page 87, much of Open Prolog's run time system is written in Prolog, and is kept private so that it can not be seen or modified by the user. Calls to private clauses are made by `privateCall` or `privateLastCall` instructions, which bypass the Name Table and reference the [first] clause of their target procedures directly. Linking private calls to their clauses is done at startup time as follows. At startup time, all calls made in clauses, whether made by `call`/`lastCall` or `privateCall`/`privateLastCall` instructions, reference their targets by name. Once all built-in predicate code has been loaded, whether host-coded or Prolog coded, all private call destinations are dereferenced through the name table and the name references are replaced by the location of the first clause having that name. Once all private calls have been linked in this manner, the names of private predicates

are removed from the name table.

Chapter 5

Compilation

5.1 Summary and Contributions

Given the straightforward connection between Prolog source code and OPAM machine code, compilation is a relatively simple process, and is implemented in procedural code. The two-phase process by which a term is read from text and converted to constructed term representation is described. The process of compiling clause code from term code is also described.

The `call/1` built-in predicate is described, where the sequence of goals to be called is compiled before execution.

The contributions in this section are the design of the tokeniser, the parser and the clause code compiler.

5.2 Introduction

In Open Prolog, terms are represented in OPAM Term Code and clauses are represented in OPAM Clause Code. Thus, for example, when the `read/1` predicate reads a term, it is actually compiling an OPAM Term Code representation of the term from its textual form. Similarly, when a clause is

Predicates	*Form of Compilation*
`read/1`	Text $\mapsto$ OPAM Term Code
`assert/1`, `findall/3`, etc.	OPAM Term Code $\mapsto$ OPAM Clause Code
`call/1`, `catch/3`, etc.	OPAM Term Code $\mapsto$ OPAM Clause Code

Table 5.1: Forms of Compilation in Open Prolog. The symbol $\mapsto$ indicates compilation, e.g. *Text* $\mapsto$ *OPAM Clause Code* means text is compiled to OPAM Clause Code.

asserted using, say, the `assert/1` predicate, a compilation from OPAM Term Code to OPAM Clause Code is performed.

OPAM code is generated in a number of different situations in Open Prolog (see also Table 5.1):

- When terms are input using the `read/1` and related predicates. Textual representations of Prolog terms are compiled into the internal representation of such terms, which is *constructed terms* of OPAM Term Code.
- When clauses are asserted, the Prolog term representing each clause is compiled into OPAM Clause Code.
- When a goal sequence is called for execution using a `call/1` or similar predicate, the Prolog term representing the goal sequence is compiled into OPAM Clause Code for execution. (This code is discarded when no longer needed.)

A separate compiler is used for built-in predicates written in Prolog (see section 6.2). This compiler was initially used to cross-compile Prolog source code into OPAM code before Open Prolog was working, but is still used to compile any built-in predicate code written in Prolog.

There are many ways to implement

In the next sections, the processes involved in compiling text to OPAM Term Code and compiling OPAM Term Code to OPAM Clause Code will be described.

5.3 Compiling Text to OPAM Term Code

This process occurs when the `read/1` predicate is called, and it consists of two phases. In the first phase, the text is tokenised and parsed using a simplified intermediate tag-based representation scheme. In the second phase, this intermediate representation is translated into constructed terms of OPAM Term Code.

5.3.1 Phase One—Text to Intermediate Representation

Text is tokenised and parsed into a structured tag-based representation by two separate assembly language routines, `readRawTokens` and `getSubterm`. The tags are listed in Table 5.2.

The routine `readRawTokens`, based on the finite state machine shown in Figure 5.1, analyses incoming text to produce a list of tagged tokens—the *Token Sequence*. The Token Sequence indicates the type, location and length of each token in the text, along with one piece of data.

The Token Sequence is provided as input to `getSubterm`, which parses the tokens into Prolog terms, taking current operator definitions into account, returning the Prolog term, still token-based. The token-base term will then be translated into constructed term or source term form as necessary.

Token Tag	*Contents of Data Field of Token*
`tokenNothing`	–
`tokenAtom`	Name Table Entry
`tokenInteger`	32-bit Unsigned Value
`tokenString`	Name Table Entry
`tokenVar`	Name Table Entry
`tokenStop`	–
`tokenPunctuation`	–
`tokenSpace`	–
`tokenEOF`	–
`tokenError`	Error Code
`tokenStructure`	Pointer to Structure Description
`tokenEndOfStructure`	–

Table 5.2: Tokeniser Tokens. These tokens are used to annotate the incoming text, classifying sequences of characters in the incoming text stream into tokens. Each token record includes the start and the length of the token as well as a piece of data as listed here. The 'Name Table Entry' for atoms, variables and strings is the location in the Name Table where the atom name, variable identifier or character sequence, respectively, is placed. A structure is represented by a `tokenStructure` token followed by a pointer to a sequence containing the structure's principal functor followed by tokens for each argument and terminated by a `tokenEndOfStructure` token.

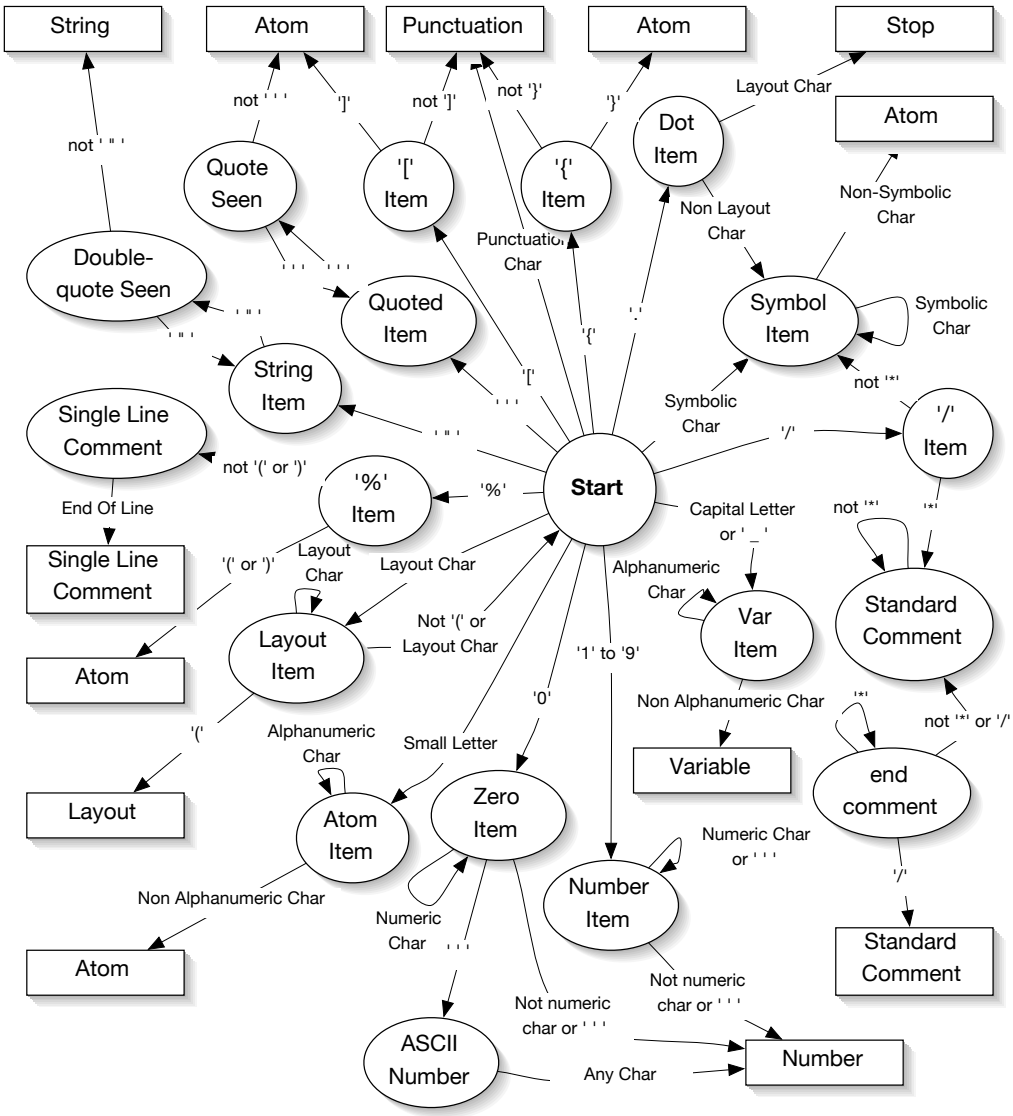

Figure 5.1: This Finite State Machine diagram outlines the design and output of the 'raw' tokeniser in Open Prolog. Ellipses ands circles represent intermediate states and rectangles represent ending states, where a token is emitted. Arrows are state transitions, labelled with the conditions for taking them. Other details like error handling arrangements are omitted for clarity.

The Parser

As previously mentioned, the parser, called `getSubterm`, takes a sequence of 'raw' tokens and produces an appropriately structured Prolog term, also expressed in a token-based notation. The Prolog language presents special difficulties with regard to parsing because it allows the programmer to define unary and binary *operators*. Any unary functor can be declared to be a prefix and/or postfix operator, and any binary functor can be declared to be an infix operator.[1] As well as the type of an operator, its precedence and associativity with respect to other operators can be defined.

Operators can be defined, undefined or modified within a file as it is being consulted, so it is clearly imperative that the parser must have immediate access to the current operator definitions at all times, and that the parser be structured so that it can deal with any legal operator definitions. A desirable characteristics of the parser is that it should give accurate error messages.

The approach taken in Open Prolog is to base the parser on a state machine which passes through a small number of states while parsing the incoming tokens and which is called recursively to handle subterms. An Operator Stack deals with operators, based on their type, precedence and associativity. The Operator Stack is also used for dealing with principal functors of regular terms. An Operand Stack and a Structure Stack are also used. The routine has eight states and uses four principal data structures and a number of other pieces of data, all of which are listed in Tables 5.3 and 5.4.

The parser uses the Operand Stack to store representations of terms, including the output of the routine and terms that, temporarily, are not part of any structure. Since it's a stack, operands are retrieved in the reverse order to that in which they are stored.

[1] It is possible in Prolog for a name to be the name of a prefix, postfix and infix operator simultaneously.

State	*Significance*
`SubtermStart`	Starting to parse a token
`SubtermPost`	After parsing a token
`SubtermFollowAtom`	After parsing an atom token
`SubtermFollowMinus`	After parsing the '-' atom token
`SubtermFollowPost`	After parsing a postfix operator
`SubtermGetCloseBracket`	After parsing a bracketed term
`SubtermGetCloseChain`	After parsing a chain-bracketed term

Table 5.3: Parser States.

Data Structure	*Usage*
Token Sequence	'Raw' tokens from Tokeniser
Operator Stack	Operators & Principal Functors
Operand Stack	Simple Terms & Structure Tokens
Structure Stack	Structures
`op_store`	Temporary store for the last operator

Table 5.4: Parser Data Structures and Data Items.

The Operator Stack accepts operator and all principal functor tokens,[2] processing them according to their arity, precedence and associativity. If an incoming token's priority is less than that of the operator at the top of the stack, the token is pushed onto the stack. Similarly, if the incoming token's priority equals that of the operator on top of the stack, and if it is left associative, the token is simply pushed onto the stack. Otherwise, the incoming operator triggers a consolidation (a *reduction*) of the Operator Stack.

The Operator Stack is reduced by removing the topmost operator and by pushing the structure corresponding to it onto the Operand Stack. The structure is actually formed in the Structure Stack by pushing a `tokenEndOfStructure` followed by the appropriate number of arguments (popped from the Operand Stack), followed by the Name Table Entry of the operator. All that remains is for a `tokenStructure` to be pushed onto the Operand Stack, with its data field pointing to the start of the structure in the Structure Stack.

When this is complete, the operator that was top of the Operator Stack is gone, converted to a structure, and another attempt is made to accept the incoming item.

Operation of the Parser The parser starts with a priority, (which, in standard Prolog, can range from 0 to 1200), and attempts to parse from the token sequence a Prolog term, whose priority does not exceed the priority given. The parser may return with a valid term or with an error. It may also return with a valid term but without reaching a `tokenStop` token.

The states and procedures the parser uses are described in the following paragraphs:

[2]The Operator Stack should perhaps be called the Functor Stack, as it handles all principal functors, whether they are operators or not. A functor that is not defined as an operator is taken to have zero precedence and no associativity.

getSubterm This is the start of the `getSubterm` routine, and a priority value is supplied as a parameter. This indicates that a term (or subterm) is sought whose priority is less than or equal to the priority supplied. An operator of this precedence, plus one, is pushed onto the Operator Stack. (Later, when the routine exits, another operator of the same precedence is pushed onto the stack, reducing all the operators remaining there after execution of the routine.)

```
int getSubterm(int in_priority) {
    int subterm_priority = in_priority // subterm_priority is the maximum priority of the term
    operator_store op_store; //used to store a token provionally identified as postfix
    int result_code = ok;
    // push_operator(<priority>,<precedence>,<name>,<arity>)
    push_operator(in_priority+1,nil,nil,0); // push this marker onto the stack to begin with...
    int state = SubtermStart;
    boolean operand_required = false; // set to true when an operand is really needed,
    // e.g. after an infix operator.
    do { // exit by a goto
        get_next_token();
        switch {state} {
```

`SubtermStart` In this state, the parser attempts to parse a term by reading a token. If the token is a `tokenVar`, a `tokenString` or a `tokenInteger`, then the parser pushes it onto the Operand Stack and transitions to the `SubtermPost` state. If the token is a `tokenPunctuation`, then if it is an open bracket or an open chain bracket, the parser recursively reads in a subterm of priority 1200, and transitions to the `SubtermGetCloseBracket` or `SubtermGetCloseChain` state respectively. If the punctuation item is an open square bracket, the parser calls the `GetList` procedure to read a (possibly empty) list and then transitions to the `SubtermPost` state. Finally, if nothing can be parsed, the parser returns a `no_token` status, unless the current token is `tokenEOF`, in which case the `unexpected_eof` status is returned.

```
case SubtermStart:
    if (the current token is a variable or string or number) { // an ordinary operand, for sure
        push_operand(current_token); // push it onto the operand stack
        next_state = SubtermPost;
```

```
    } elseif (current_token=='(') { // a bracketed subterm
        if (getSubterm(1200)==no_token) {
            goto LabelSubtermBadCharError;
        }
        next_state = SubtermGetCloseBracket;
    } elseif  (current_token=='{') { // a chain-bracketed subterm
        if (getSubterm(1200)==no_token) {
            goto LabelSubtermBadCharError;
        }
        next_state = SubtermGetCloseChain;
    } elseif (current_token=='[') { // a list, maybe empty
        if (GetList()!=ok) {
            goto LabelSubtermErrorExit;
        }
        next_state = SubtermPost;
    } elseif (current_token is an atom) {
        if (current_token=='-') {
            next_state = SubtermFollowMinusAtom; // if it's followed by a number, it's special
        } else {
            next_state = SubtermFollowAtom; // it's an atom, but it could be an operator
        }
    } else {
        // nothing parsed, but check for EOF
        if (current_token is the EOF token) {
            backup_token(); //return EOF token as unused
            result_code = unexpected_eof;
            goto LabelSubtermExit;
        } else {
            backup_token(); //return the token unused
            result_code = no_token;
            goto LabelSubtermExit;
        }
    } else {
        goto LabelSubtermBackupAndExit; // nothing parsed, but that's ok.
    }
break;
```

`SubtermPost` Here, the parser has already seen what could be a complete term, reads the next token, and looks for a `tokenStop`, a postfix operator, an infix operator or a comma (treated as an infix operator of priority 1000). If the token is an operator, its priority must not be greater than that of the subterm as a whole; if it is, then it is not consumed: instead, the token is returned to the start of the Token Sequence and the tokeniser exits with the term already parsed as a result.

There is always the possibility that the token could be defined both as a postfix operator and as an infix operator. In that case, if the priority of

the postfix operator is suitable, the parser provisionally accepts the postfix interpretation and defers a final choice until the next token is read in the `SubtermFollowPost` state. Since the token is only provisionally accepted as postfix, it is premature to apply it to the operator stack. Accordingly, the token is stored temporarily in the `op_store` where it may be accessed in the `SubtermFollowPost` state.

```
case SubtermPost:
    if (the current token is the stop token) {
        goto LabelSubtermBackupAndExit;
    } elseif (current_token = ',') {
        if (subterm_priority>1000) { // the subterm we're asking for must have
        // a priority > 1000 to include an infix comma
            push_operator(1000,xfy,',',2);
            operand_required = true;
            next_state = SubtermStart;
        } else {
            goto LabelSubtermBackupAndExit;
        }
    } elseif (
      (current_token is defined as a postfix operator) &&
 (its postfix priority < subterm_priority)
        ) {
        // here, an atom defined as a postifx operator is in the right place
        // with an appropriate priority
        // but it might still 'really' be an infix operator.
        // We note it's existence as a valid postfix op
        // and wait for the next token, in SubtermFollowPost
        // to decide finally if it's posfix or infix.
        // We prefer infix by virtue of the eager consumer rule.
        op_store = current_token; // store priority, precedence, name and arity for later
        next_state = SubtermFollowPost;
    } elseif (current_token is an infix operator && (its infix priority < st_priority) {
        with current_token.infix_properties
            push_operator(priority,precedence,name,2); // this may reduce the operator stack
        operand_required = true;
        next_state = SubtermStart;
    } else {
        goto LabelSubtermBackupAndExit;
    }
break;
```

`SubtermFollowAtom` Having read a `tokenAtom`, the next token is read to determine whether the `tokenAtom` is the principal functor of a structured term written in standard form or something else. If the token is followed immediately—without white space—by an open round bracket, it is taken to be the principal functor of a structured term, and the routine `getArguments`

is called to read the arguments onto the Operand Stack. Following this, an operator of zero priority, having an arity equal to the number of arguments returned by `getArguments`, and with a name equal to the name of the `tokenAtom`, is pushed onto the Operator Stack. The structured term is actually constructed when the stack is next reduced.

If the token following the `tokenAtom` is not an open round bracket, it is not consumed, but is pushed back into the start of the Token Sequence. Consideration returns to the original token. If the original token is not a prefix operator, then it is simply pushed onto the Operand Stack and the machine transitions to the `SubtermPost` state. If it is a prefix operator, then if its priority is not greater than that of the subterm sought, it is pushed onto the Operator Stack and a recursive call to `getSubterm` is made, with a priority equal to that of the operator. If the procedure returns normally, then the state machine transitions to the `SubtermPost` state.

It is legitimate in Prolog to have a prefix operator without an argument, so it is not an error for the call to `getSubterm` to return with the `no_tokens` status. Consequently, in the event of a `no_tokens` status being returned, the prefix operator is removed from the operator stack, and instead the `tokenAtom` itself is pushed onto the Operand Stack.

```
case SubtermFollowAtom:
    next_state=SubtermPost; // continuation
    if (current_token = '(') { // an atom followed by an open bracket is a functor
        GetArgumentsAndArity();
        // push the principal functor onto the operator stack with a precedence of zero
        // and an arity equal to the number of arguments. It will be reduced later.
        push_operator(0,nil,previous_token name,arity);
     } else {
        // here, the situation is an atom and its following token have been read,
        // but the following token is not a '('
        // We now go back to see if the atom was a prefix operator
        backup_token(); //return the token read after the atom
        if (current_token is a prefix operator) {
            if (current_token prefix op priority > subterm_priority) {
            // priority too high for subterm requested
                goto LabelSubtermBackupAndExit;
            } else {
                // commit to the fact that the atom was a prefix operator
                operand_required = true;
```

```
                with current_token.prefix_properties
                    push_operator(priority,precedence,name,1);
                if (getSubterm(current_token.prefix_properties.priority)==no_token) {
                // i.e. if we have a prefix operator and its argument
                    // it turns out that the prefix operator has no arguments.
                    // we have already used it to possibly reduce the operator stack,
                    // which is legal,
                    // but we don't want it to consume any later tokens on reduction
                    // so we remove it from the operator stack and push it on as an atom
                    pop_operator();
                    push_current_operand();
                }
            }
        } else {
            push_current_operand(); // it was an ordinary atom
        }
    }
break;
```

`SubtermFollowMinus` The next token is read. If it is a `tokenInteger`, then a `tokenInteger` operand is pushed onto the Operand Stack with a value of the negative of the `tokenInteger` token just read from the Token Sequence. The machine then transitions to the `SubtermPost` state. Otherwise, the machine executes the
`SubtermFollowAtom` code.

```
case SubtermFollowMinus:
    // if the minus sign is followed by an integer, take it as a negative integer;
    // otherwise, treat the minus sign as an atom
    if (current_token is an integer) {
        if (range is ok) {
            push_operand(negative of value of current_operand);
            // the negative integer is now treated the same as a regular integer, var or string
            next_state = SubtermPost;
        } else {
            result = range_error;
            next_state = SubtermExit;
        }
    } else {
        backup_token(); // push back the token, now we know it's not an integer
        next_state = SubtermFollowAtom; // treat the minus sign as an atom
    }
break;
```

`SubtermFollowPost` This state is entered to support or refute the tentative identification of a `tokenAtom` in the previous state as a postfix operator. That atom is stored in `op_store`. The next token is read. If it can be parsed as

part of the term, then the tentative identification of the previous atom as a postfix operator is confirmed; otherwise, the identification of the previous token as a postfix operator is refuted, and the token just read is returned to the start of the token sequence.

If the token just read is either a `tokenStop`, a token that could be another postfix operator, or an infix operator (including a comma) of appropriate priority, the previously identified postfix operator itself is copied from the temporary `op_store`—where it had been placed during the previous `SubtermPost` state—and pushed onto the operator stack.

Next, if the token is `tokenStop`, the routine exits; if the token is a postfix operator, the machine executes the `SubtermPost` code; if the token is an infix operator, the operator is pushed onto the operator stack and the machine transitions to the `SubtermStart` state.

```
case SubtermFollowPost:
    // here, a postfix operator has been seen and provisionally accepted.
    // It might turn out to be an infix operator
    // which is preferred. We look at the next token to see.
    // The next token could be a stop, a comma, another postfix operator,
    // an infix operator, or none of them
     if (current_token is a 'stop') {
        goto LabelSubtermFollowPostBackupAndExit; // yep, it was a postfix op. Take it and go.
    } elseif (current_token==',') {
        if (op_store.priority>999) { // if it's a postfix operator then its priority is too high
             goto LabelSubtermFollowPostBackupAndExit; // can only be an error --
             // Accept postfix and complain about the comma
        } else {
            with op_store // accept the previous token as postfix...
                push_operator(priority,precedence,name,1);
            push_operator(1000,xfy,',',2); // push the comma
            operand_required = true;
            next_state = SubtermStart; // continue
        }
    } elseif (
                (current_token is a postfix operator) &&
                (current_token.priority>op_store.priority) && // is greather than previous
                (subterm_priority>current_token.postfix_properties.priority) // is low enough
                // to include in current subterm
             ) {
        with op_store // accept the previous token as postfix...
            push_operator(priority,precedence,name,1);
        backup_token();
        next_state = SubtermPost; // reconsider the current token on the next state transition
    } elseif (
                (current_token is an infix operator) &&
```

```
                (current_token.priority<subterm_priority) &&
                (current_token.priority>op_store.priority) // a new postfix operator,
                // of higher priority than the stored one
                // follows it, so commit to interpreting the stored one as a postfix operator
             ) {
        with op_store // commit to the previous token as postfix...
            push_operator(priority,precedence,name,1);
        with current_token.infix_properties // commit to the current token as infix
            push_operator(priority,precedence,name,1);
        next_state = SubtermStart; // reconsider the current token
    } else {
    // getting here means that is was not possible to sustain the idea that
    // the previous token was a postfix operator
    // because the token following it (i.e. the current token) could not be accommodated
    // -- either it was the wrong kind, or the not a suitable postfix or infix operator.
    // Therefore, we reconsider the previous token now to see if it was infix.
    // If it's not infix, it's a syntax error
    // If it is infix and its priority is too high, we just stop parsing before it --
    // it's not an error we can deal with here.
    backup_token(); // get back to the previous token
    if (current_token is an infix operator) {
        if (current_token.infix_properties.priority<subterm_priority) {
            with current_token.infix_properties
                push_operator(priority,precedence,name,2);
            next_state = SubtermStart;
        } else {
            goto LabelSubtermBackupAndExit; // can't use it because its priority is too high
        }
    } else {
        // if it's not infix, it's at error
        goto LabelSubtermBackupAndError;
    }
break;
```

`SubtermGetCloseBracket` Having recursively called `getSubterm` to read a Prolog term of priority 1200, the machine enters this state and reads the next token. If it is a close round bracket punctuation token, the machine transitions to the `SubtermPost` state; otherwise it exits with an `unexpected_char` status.

```
case SubtermGetCloseBracket:
     if (current_token==')') {
        next_state = SubtermPost; // accept the bracketed term and continue
    } else {
        backup_token();
        result = unexpected_char;
        next_state = SubtermExit;
    }
break;
```

`SubtermGetCloseChain` Having recursively called `getSubterm` to read a Prolog term of priority 1200, the machine enters this state and reads the next token. If the token turns out to be a close chain bracket, a structured term whose principal functor is `{}/1` is composed by pushing an operator of name {}, with zero priority and an arity of 1 onto the Operator Stack. (The structure is actually composed when the Operator Stack is reduced.)
If the token is not a close chain bracket punctuation token, the machine exits with an `unexpected_char` status.

```
case SubtermGetCloseChain:
     if (current_token=='}') {
        push_operator(0,nil,'{}',1); // to compose the chain-bracketed unary term
        next_state = SubtermPost;            // accept the chain-bracketed term and continue
    } else {
        backup_token();
        result = unexpected_char;
        next_state = SubtermExit;
    }
break;
```

End There remains the last part of the parser that is accessed on exit from the state machine.

```
     } // end switch
 state = new_state;
 } while (true); // loop until it jumps out

// these are all destinations of branches inside the switch. They exit the while loop too.

// Normal Exit Points
label LabelSubtermFollowPostBackupAndExit:
    // we want to exit, but there's a provisionally identified postfix operator in the op_store
    // push it onto the operator stack and exit...
label LabelSubtermBackupAndExit:
    backup_token();
label LabelSubtermExit:
    // push on this operator to reduce the operator stack
    push_operator(subterm_priority+1,nil,nil,nil);
     // remove the starting and ending operator -- stack is back to original state
    (operator_stack++)++;
    return result_code;

// Error Exit Points
label LabelSubtermBackupError:
    backup_token();
label LabelSubtermBadCharError:
    result_code = unexpected_character;
```

```
    clean up operator stack by removing all operators up to
    and including the marker operator
    placed at the start of the call to getSubterm

    return result_code;
}
```

`GetList` Having seen an open square bracket in the state `SubtermStart`, this procedure is called to read the rest of a (possibly empty) list.

5.3.2 Phase Two—Intermediate Representation to OPAM Term Code

Once the text has been parsed into a term in intermediate representation, it must be transliterated into an OPAM constructed term. This is facilitated by the correspondence between the way structures are referenced out-of-line in both representations. Transliteration is simply a matter of traversing the intermediate representation of the term, converting each item into an equivalent constructed term, recursively generating molecules for the structures encountered in the term. The other task performed in phase two is to alias identically-named variables in the parsed term to the same OPAM variable. The output from phase two is therefore a constructed term version of the term and a list of the variable to variable-name pairs.

5.4 Compiling Prolog to OPAM Clause Code

OPAM clause code is generated by separate programs in three situations:

- When a `call/1` is executed, the goal sequence represented by the argument is compiled into OPAM code for execution. A special feature of this situation is that the code is discarded when no longer needed;

- When a clause is asserted, say by the `asserta/1`, `assertz/1` or `findall/3` predicates.
- When built-in predicates are being compiled into PRLC resources using a separate compiler;

The process is essentially the same in all cases, and consists of three phases. In the first phase, like the first pass of an assembler, the size of the code and a variable table is built up; next, the variable table is analysed to calculate the classification and ordering of the variables; then, in the third phase, the source code is actually generated.

In the following sections, these processes, by which a Prolog term (in OPAM Term Code) is compiled to a Prolog clause (in OPAM Source Code), are described.

5.4.1 Phase One—First Pass

Just as in an ordinary assembler, in phase one the source (a Prolog term in OPAM Term code) is traversed to determine the maximum size of the output code. In addition, each time a variable is encountered, information about it is entered into a Variable Table, as listed in Table 5.5.

5.4.2 Phase Two—Variable Analysis

Variables are analysed into `local`, `global` or `void` according to the following rules, taken in order:

1. If a variable is used only once, and is not used in a structure, then it is a void;
2. If a variable is used more than once, does not occur in a structure and is not used in the body, then it is a temporary, which will be encoded as a local;

Item	*Significance*
`identity`	The location of this variable, used to identify it uniquely.
`usage`	Number of times the variable is used.
`firstUse`	Usage flags detailing where it was first used.
`allUses`	Usage flags detailing all circumstances of its use.
`firstUseCode`	Source Term code to use for first use of variable
`restUseCode`	Source Term code to use in subsequent uses.
`firstUseCodeSize`	Length of first use code
`restUseCodeSize`	Length of subsequent use code

Table 5.5: Variable Table Entry. The Variable Table is constructed during Pass 1. Information about a variable's first and subsequent uses is recorded. After Pass 1, the kind of variable is determined, and the codes used for the first and subsequent uses are recorded in the table for use during Pass 2. Table 5.6 details the usage flags used in the `firstUse` and `allUses` fields.

Flag	*Significance*
`varHeadUse`	The variable is used in the head of the clause.
`varGoalUse`	The variable is used in a goal.
`varLastGoalUse`	The variable is used in the last goal.
`varStructUse`	The variable is used in a structure.
`varEnumerated`	The variable has been assigned a location (used during code generation).
`varUsed`	The first occurrence of the variable has already been encoded (used during code generation).
`firstUseCodeSize`	Length of first use code
`restUseCodeSize`	Length of subsequent use code

Table 5.6: Variable Usage Flags. The `firstUse` and `allUses` entries in the Variable Entry Table (Table 5.5) use these flags to register the types of usage of the variable.

3. If a variable is used more than once, and does not occur in a structure nor in the last goal, then it is a local;

4. Otherwise it is a global variable.

The number of variables of each kind that must be initialised before use is calculated. Globals and locals that appear first in a head and not in a structure do not need initialisation, as they will receive values when they are first used.

Once variables have been classified, their relative positions within their frames are determined, allowing for whether they must be explicitly initialised before use. Variables that need initialisation are numbered so that they occupy the bottom of each frame.

The actual numbering of each variable is actually its offset from the bottom of its frame. Global variables begin eight bytes up from the bottom of the global frame, and local variables begin forty-eight bytes up from the bottom of the local frame. Each variable occupies eight bytes.

- Voids have no numbering;
- Globals needing initialisation are numbered from 8 upwards, in steps of 8, in order of appearance;
- The remaining globals are numbered in order of appearance, starting above the highest global that must be initialised;
- Locals needing initialisation are numbered from 48 upwards, in steps of 8, in order of appearance;
- The remaining locals are numbered in order of appearance, starting above the highest local that must be initialised;

5.4.3 Phase Three—Code Generation

At the start of this phase, the variables have been classified and the maximum size of the output code has been calculated. Taking advantage of the latter, a memory block is allocated to accommodate the resulting code. In an arrangement somewhat reminiscent of the organisation of an application zone into a heap and a stack, this memory block is organised into a *fragment space* at the low end, growing upwards, and a *code space* at the high end, growing downwards. Briefly, code is *generated* in the fragment space but *stored* in the code space. This arrangement allows the generation of code in a recursive fashion to facilitate the generation of code for nested structures.

Code generation proceeds as follows:

- The number of globals needing initialisation, less 1, is written to the fragment space;
- The code for each argument, if any, of the clause head is output:
  - If the argument is a simple term such as an atom or integer, its code is placed sequentially in fragment space.
  - If the argument is a variable, then the Variable Table is consulted to determine the correct code to emit. Again, its code is placed sequentially in fragment space.
  - If the argument is a structure, two distinct sets of code must be emitted: the code for the structure itself and a `structure` token followed by an offset referencing the structure.

    The structure code is generated as described below, and is placed at the upper end of the memory block, in code space. Once placed, its location is known, so the `structure` token and the offset can now be emitted in fragment space. In this way, fragment space grows upwards and code space grows downwards.

- The code for the neck structure is emitted in fragment space.
- The code for each goal, if any, is emitted. Again, if any argument is a structure, it is constructed in code space, and a reference to it is placed in fragment space.
- The code for the foot structure, if necessary, is emitted.
- Any remaining space between the end of the foot code and the start of the structure code is closed up by moving the structure code downwards.[3]

Recursive Generation of Structure Code

A structure is generated using the fragment space pointer and the code space pointer:

- The functor table entry for the principal functor is emitted in fragment space;
- Each argument of the structure is emitted in fragment space. If it is a simple term, the appropriate code is emitted; if it is a variable, the Variable Entry Table is consulted. If it is a structure, the structure code is generated in code space by a recursive call, and a `structure` (or `structureLand`) tag followed by the offset to the location of the new structure code in code space is emitted in fragment space.
- When the structure code has been fully emitted (in fragment space), it is copied up to the code space, freeing the fragment space and occupying the top of the code space stack where it can be referenced from code still in fragment space.

[3]This is not really necessary, as all that will be saved is space due to the replacement of regular variables with voids, and is probably quite an expensive operation.

It can be seen that code is generated in fragment space but is moved to code space when finalised. In this sense, the memory block fills from the top down.

5.4.4 The `call/1` predicate

The other place that a form of OPAM Source Code is generated is in the `call/1` predicate: the goal or goals to be called are compiled into OPAM Source Code for evaluation. The alternative approach of implementing `call/1` with a Prolog metainterpreter was discarded since it departed from the idea of having just one mode of operation. Secondly, a significant amount of work would be needed to get the operation of the metainterpreter to match that of the system itself, particularly in regard to negation, the cut and the control constructs. Thirdly, given the relative ease with which compilation can be done, it was overall considered best to compile and then the call argument.

This is different from the general case because the clause is formed from a number of goals and has no head. Moreover, since the code is used just once, it is argued that the classification of variables into local and global is not important. A further feature of this code is that it is not placed in the OPAM Code Space—in view of its transient existence, it is placed in a local stack frame, which can be deallocated readily as soon as the code is no longer needed. This makes it necessary to ensure that no future molecule can be comprised of any structures within the 'temporary' call code. This is done by ensuring that all structures defined in the call are realised as constructed terms.

The `call/1` predicate is implemented using a combination of Prolog and assembly language predicates, as shown in the following code:

```
call(Term) :-
    assembleGoalSequence(Term,
    IntermediateCode-[foot,clause_dont_mark|DeferredCode],
```

```
DeferredCode-[],0),
computeOffsets(IntermediateCode,Ic2,0,Size),
resolveOffsets(Ic2,Code),!,
'new$system$call'([Size|Code]).
```

The term to be called is assumed to be a sequence of goals, and is assembled in one pass by `assembleGoalSequence/4`.[4] This takes the term `Term` and the location counter to produce two difference lists, the first of which is the code for the goals, and the second of which is all the deferred code, i.e. the call sequences and nested structures. In turn, `assembleGoalSequence/4` calls a predicate called `assemble/4` to generate code for each item in the goal sequence.

For example, the goal sequence

```
a(X),!,b(c,d)
```

yields the joined difference list:

```
[call,offset(A),cut,call,offset(B),foot,clause_dont_mark,
 label(A),callFrame,a(C),label(B),callFrame,b(c,d)]
```

The locations of the labels in the difference list are calculated using `computeOffsets/3`, the offset elements replaced with word elements and the labels removed.

This yields the following for our example:

```
[call,word(14-2),cut,call,word(24-8),foot,clause_dont_mark,
 callFrame,a(C),callFrame,b(c,d)]
```

Finally, the list is converted into a list of the numeric codes using `equ` statements (not shown) to equate names with the code numbers corresponding to each instruction, with the call sequences at the end.

This gives the following result:

[4]The code for the predicates described here is listed on page 237.

```
[4,12,28,4,16,32,68,a(C),b(c,d)]
```

The size of the code is prepended to this and the entire list is passed to `'new$system$call'` for loading and execution.

All the body code is assembled by these simple routines, and the predicate copies these codes into space it reserves in the local stack. For each atom or structure designating a call sequence at the end of the list, the predicate constructs call sequence code comprising the functor name followed by a global variable for each argument, each unified with the corresponding subterm in the list.

General compilation is organised along the same lines as compilation of `call/1` arguments, but with the added complexity of variable analysis, clause head argument generation, neck and foot code generation and finally the generation of a clause descriptor record and the installation of the code in the code space. The compilation of clauses being asserted is done in one assembly language program called `sourceAssemble`.

Separately, a cross-compiler, `ecompile/1`, or the compilation of built-in predicates into `PRLC` resource code was written entirely in Prolog. The OPAM object code output is stored in a text file format called `S-Records`. A simple Macintosh-specific utility program converts `S-Record` object files into resource files containing the `PRLC` resources.

Chapter 6

Built In Predicates

6.1 Summary and Contributions

Five different schemes for implementing built-in predicates are described. Modules of Prolog code are separately compiled and stored as Macintosh OS 'resources'. The 'standard' procedural built-in predicate arrangements are described and a considerably faster but more restrictive scheme ('fast predicates') is described. Arrangements for plug-in built-in predicates ('external predicates') are also described, including the ability to deal with ersatz interrupts. Finally, the idea of input/output 'streams' is introduced as an extension to external predicates.

Perhaps of most interest here is how external predicates are organised.

6.2 Introduction

Prolog incorporates built-in predicates serving a range of purposes. For example, the built-in predicate `write/1` is a built-in predicate that always succeeds when called, but which has the [desired] side-effect of outputting a textual representation of the term supplied to it as an argument. Similarly,

the built-in predicate `read/1` has the side-effect of inputting sufficient characters to represent a Prolog term. If this term unifies with the predicate's single argument, then the predicate succeeds; otherwise, it fails. Whether it succeeds or fails, the side effect of 'consuming' characters from the input remains unchanged. Other built-in predicates exist to perform arithmetic, to form new terms and to extract arguments of terms, to do comparisons, to examine and set environmental flags, and so on. Some built-in predicates can be written in Prolog, and it is necessary to hide the Prolog code used to implement them from the user. Many built-in predicates are entirely procedural, and are most conveniently written in a conventional procedural language. And of course, a considerable number of built-in predicates are built from a combination of Prolog and procedural code. In these cases, the procedural code is built in to private built-in predicates that are in turn used by the Prolog code to implement the built-in predicates in question.

Reflecting this, and reflecting the experience gained as Open Prolog was implemented, built-in predicates are implemented in five ways in Prolog:

- Modules of Prolog code
- 'Standard' Predicates
- 'Fast' Predicates
- External Predicates
- Streams

Built-in predicates of first category above—modules of Prolog code—are written in normal Prolog; they are discussed in Section 6.3 on page 150 below. The other four kinds of built-in predicates are implemented in 68000 code, and are described in the later sections of this chapter. The basic ideas for these four types are simple. *Standard* predicates are implemented as hybrid clauses where the OPAM code for a Prolog clause head is followed by

a 68000 routine rather than OPAM code for the clause body. This is done by replacing the `neck` by an OPAM instruction to switch to host processor instruction execution. In many cases, the calling overhead of this arrangement was high relative to the amount of useful processing, so the *fast* built-in predicate schema transfers execution to host processor almost as soon as the procedure is called, bypassing most of the standard predicate's overhead. In an effort to facilitate development of procedural predicates without having to have knowledge of the low-level architecture of Open Prolog, *external predicates* allow the development of predicates to interact with Open Prolog through a shared parameter block and a well-defined set of commands and protocols. These can conveniently be written in a high-level language like C++. The calling overhead for external predicates is very high, however, so the final built-in predicate schema, *streams*, is an extension of external predicates that seeks to confine the high overhead to the setting up and tearing down of data streams such as files, while the actual transfer of data is fast.

Open Prolog uses `PRLC` resources to hold built-in predicates implemented in Prolog and encoded in OPAM code. Standard and fast predicates have an OPAM code stub as well as the actual 68000 code itself. The OPAM stub code is stored in `PRLC` resources and the 68000 code is stored in regular `CODE` resources. External predicates have an OPAM stub generated dynamically at system startup time; the 68000 code is stored in `PRLX` resources.

6.3 Modules of Prolog Code

Many built-in predicates are themselves written in Prolog. These procedures, having been compiled into OPAM instructions, are stored, along with link data, in `PRLC` resources in the resource fork of Open Prolog or in files stored in the Open Prolog Additions folder. At startup, the contents of these resources

are copied into the Code Space and linked together.[1]

The modularity of `PRLC` resources enables modules of built-in predicates to be added easily to Open Prolog. A special Prolog compiler and resource formatter is used for building Prolog code modules. ResEdit is used to transfer the resulting `PRLC` resources into the Open Prolog resource fork.

6.4 Standard Predicates

The first of the four schemes for dealing with procedural 68000 code is straightforward — a standard predicate is called in the normal way, and the procedure's head code is normal OPAM code, where each argument is a local variable. Thus, when a standard predicate is called, a new local frame is partly constructed for the predicate. However, instead of the head code being followed by a `neck` instruction, as would be the case for a clause written in Prolog, it is followed by a `proc` instruction, which begins execution of 68000 code at the location immediately following. This code fully dereferences the argument in each local variable, then transfers program execution to the 68000 code that forms the body of the procedure, located in a `CODE` resource. Standard predicates can be determinate or non-determinate, and upon exit, the appropriate `procedureExit` or `indetProcedureExit` OPAM instruction is executed to resume OPAM operation.

Originally, this was the only way in which 68000 procedures could be written. Although it is a general solution, problems include its high overhead and its tight connection with the Open Prolog runtime environment:

- The high overhead is because a full call action procedure[2] is executed when the procedure is called, which checks for first argument matching

[1] In the present release of Open Prolog, there are eighteen `PRLC` resources. The largest resource contains a large number of built-in predicates and also contains the familiar interactive command-line 'listener'.

[2] Call instructions and call action procedures are discussed on page 104

and alternative clauses—almost always unnecessary for built-in procedural predicates. In addition, a new local stack frame is partly constructed for the predicate, and the goal arguments are unified with local variables and then fully dereferenced before the procedure's 68000 code begins to execute. So while these operations certainly do no harm, and make programming a little easier, the time overhead they introduce is excessive; for some simple determinate procedures, the overhead is greater than the 'useful' execution time.

- The tight connection with the Open Prolog runtime meant that writing procedures was in many situations unnecessarily difficult, and essentially impossible in a high level language.

The next two schemes were designed to overcome these deficiencies, the first by minimising the calling overhead, and the second by defining a standard interface so that predicates could be written in a high level language, without exposure to Open Prolog's runtime structures.

6.5 Fast Predicates

Fast predicates dispense with almost all of the calling overhead incurred by standard predicates. Recall that when a standard predicate is called, a normal call action procedure is invoked by the call instruction. This is responsible for checking for first argument matching and for alternative clauses. In a fast predicate, the predicate code itself becomes the call action procedure. In this way, the minimum possible overhead for a call is encountered, and control passes to the 'useful' code of the predicate with greatly reduced overhead. Since a fast predicate has no local frame of its own, arguments must be accessed and dereferenced in the caller's environment. In the present version of Open Prolog, most determinate predicates are implemented as fast

predicates.

6.6 External Predicates

The third scheme—'External Predicates'—uses The Macintosh OS's implementation of resources to implement 'plug-in' predicates. External predicate source files, together with a number of interface files, are compiled into `PRLX` resources that are then added to the Open Prolog resource fork itself, or placed in files in the Open Prolog Additions folder which is searched at startup. Predicate code is activated by transferring program execution to the start of the resource, having pushed the address of a parameter block onto the stack.

The parameter block is the conduit through which an external predicate communicates with Open Prolog. A command is placed in the parameter block when a resource is activated. The commands available are:

`GetPRLXInfo` This returns the number of predicates defined in this `PRLX` resource. This is used at system startup, when the system identifies all the predicates available. In all subsequent commands, the predicate to which the command is addressed is specified with a one-based index number.

`InitialisePredicate` The code associated with the indexed predicate is given this opportunity to initialise itself. Three important parameters are supplied. The first is the address of a routine that can be used to make an interrupt request (see section 6.6.1). The second is a parameter block longword whose value will be available every time the predicate is called in the future. Typically during initialisation, a block of storage could be reserved and its pointer or handle stored in the parameter block so that it can be accessed whenever that predicate is called. The

third is another parameter block longword whose value will be available to every predicate in the `PRLX` resource so that all the predicates in the resource could have access to shared data. The `InitialisePredicate` command is issued to the predicates in a resource in index order, so that the first predicate could initialise the shared longword on behalf of all predicates.

`CallPredicate` This command is issued when the predicate is called in the course of executing a Prolog program. When this command is issued, the private longword and the shared longword are guaranteed to be available, enabling the predicate to access private and shared data.

Callback commands or *callbacks* enable the predicate code to access and modify its arguments and to interact with the Open Prolog system. The system itself is suspended during this time, thus callbacks have access to the suspended 'snapshot' of the system.

Variables may be unified with other Prolog terms, with numbers, with atoms (specified by a character string) or with structures, where the principal functor is specified by a character string and number, and where the subterms are uninstantiated variables. The subterms can be accessed using further callbacks. By this means, complex structures can be built, and terms and subterms accessed and modified by the predicate code, without having direct access to the runtime structures.

On completion, the predicate sets a return code indicating success, failure, an ISO Prolog error or an `unclassified_error` or a `user_interrupt`. If an error is returned, it is accompanied by a host error code, an argument number and a message string.

`ClosePredicate` This command has not been implemented. The intention is to issue this command just before Open Prolog shuts down to give

the predicate an opportunity to terminate cleanly.

`GetEventsVersion` This command returns the version number of the interface that the external predicate was compiled with. If this doesn't match the interface version number of Open Prolog, the external predicate is disabled.

`PRLX` resources are loaded at startup, and the predicates they contain are identified and initialised at that time. These predicates cannot be retracted or removed.

The overhead in executing external predicates can be high. When an external predicate is executed, the resource containing the predicate may have to be loaded from disk. Additionally, getting and setting arguments using callbacks is certain to be slower than accessing them directly as a normal built-in predicate would. A further disadvantage is that, since the application itself is suspended while the external procedure is running, its user interface receives no processor time, causing it to appear to freeze if the external predicate takes too long to return. This is partly due to the architecture of the Mac OS for which Open Prolog was developed, in that the Mac OS does not support threading or preemptive multitasking at the application level.

6.6.1 Events and Interrupt Handlers

While the Open Prolog application is running, the user interface code is capable of receiving 'events' from the Mac OS. Events are data structures containing information about recent user interface events, such as a mouse click, a window moving and so on. In addition, other Mac OS messages (including inter-process messages) can be made available to the application as events. One could imagine an external predicate with two components: the predicate code itself, callable from Open Prolog and returning relatively

quickly, and a separate asynchronous event handler which communicates back to the predicate code via a shared data structure such as a simple flag. In this scenario, the asynchronous event handler code could set the shared flag to indicate to the predicate code that an event had happened. However, the predicate would need to be activated by being called by Open Prolog to detect the flag. To avoid having to poll for such changes, an ersatz interrupt mechanism is implemented.

The Interrupt Handling Mechanism

When an external predicate is initialised at startup, the address of an interrupt requester is provided. When called, the requester puts a prioritised interrupt request in a global interrupt request data structure. The priority is between 1 and 7, and associated with the request is one atom. Interrupt requests are examined each time the user interface code is run, and, if the priority of a request is high enough, an `interruptRequested` flag is set. The next time the juggle code (section 4.10.2) runs the user interface, this flag is reset. If it was already set, then, rather than return to call the predicate that was about to be called when the juggle code was run, the special interrupt handler `system$start$interrupt$handler/2` is called which retrieves the atom that was supplied by the interrupt requester and the current interrupt priority level. It then sets the interrupt priority level to that of the interrupt request to block other interrupt requests of the same or lower priority. The atom can then be used as a selector for a Prolog procedure to actually service the interrupt request.

This mechanism is used, for example, for the Apple Event handler predicate `apple_event/2`. Upon initialisation, the predicate code installs an Apple Event handler in the host operating system's event handling system. When an Apple Event is received for Open Prolog, the predicate's Apple Event handler is activated. This stores a reference to the Apple Event and sends

an interrupt request to Open Prolog. The interrupt handler should then execute `apple_event/2` to get access to the input stream of text and to an output stream in which to write a reply.

6.7 Streams

The fourth scheme—'Streams'—is an extension of External Predicates to deal with plug-in Input/Output Handlers. The idea here is to have Input/Output Handlers (*IO Handlers* in the sequel) dedicated to different types of data sources and sinks, such as files, blocks of memory, dialog boxes, Apple Events etc.

Whenever a connection is made to a data source or sink, the handler that deals with that type of source or sink is associated with a descriptor that becomes the connection's ISO Prolog stream identifier. Requests for data transfer are then made directly to the handler through the connection descriptor. This scheme offers speed and flexibility; speed because when data is required, the appropriate handler method can be called immediately, and flexibility because the special needs of a source or sink can be catered for with dedicated extra external predicates.

These handlers are contained in PRLX resources along with associated external predicates. At startup, the handlers are registered by an associated external predicate, and a unique register four-character code is registered for each handler.

Whenever a connection is opened to a data source or sink, a descriptor of the connection is registered in the IO Registry, and the code of the hander for that particular stream is stored as part of the connection descriptor. The ID of the connection descriptor becomes the stream identifier to be used by ISO Prolog-compliant input/output predicates.

External Predicates have the ability to register IO Handler objects. These

handlers can then be called to perform text-based Input/Ouput through a fast interface, independent of the PRLX interface. Further, by assigning a tokeniser procedure and table to the object, input could be in the form of low-level tokens. At this time, the tokeniser functionality has not been implemented, so the IO Handler can only return sequences of characters. Consequently, IO Handlers are not fully integrated into Open Prolog. A finite state machine-based tokeniser has been developed and is used for experimental purposes.

Chapter 7

Control Constructs and Exceptions

7.1 Summary and Contributions

This chapter describes how the control constructs such as disjunction, negation, if-then, if-then-else and the `catch/3` predicate are implemented. Catch-and-throw error handling is also described in this chapter. A novelty of the design is the use of special purpose non-removable local stack frames to control the operation of the constructs and the use of linked lists of non-removable frames for efficiency. A linked list of catch-and-throw destination pointers is also maintained.

7.2 Introduction

In the typical Prolog clause body, goals are separated by commas, indicating conjunction, or possibly by semicolons, indication disjunction. Negations are represented by the `not`, `fail_if` or `\+` symbols. To this set must be added a small number of additional *control constructs*. The control constructs in ISO

Prolog are disjunction, conjunction, negation, *if-then* and *if-then-else*. Finally, most Prolog implementations now have—and ISO Prolog stipulates—catch-and-throw error handling.

Given that a close correspondence is to be maintained between the Prolog source code and the OPAM code, it follows that transformations of these constructs that alter the structure of the code are not permissible. Instead, a representation of the control constructs must allow them to be faithfully decompiled, and at the same time, the implementation must comply with the sometimes intricate semantics of the constructs. Open Prolog uses a novel and flexible scheme to represent the constructs and to implement their semantics faithfully.

While the ISO Prolog standard [39] now defines the semantics of these constructs fully, there was considerable variation in the semantics and performance of early implementations of these constructs [52]. Consider, for example, the following program:

```
a :- b,(c,!;d),e.
b :- write(b1).    b :- write(b2).
c :- write(c1).    c :- write(c2).
d :- write(d1).    d :- write(d2).
e :- write(e),nl.
```

Some implementations used a built-in predicate to implement disjunction, as if disjunction was defined as follows:

```
(X;Y) :- call(X).
(X;Y) :- call(Y).
```

Two immediate issues arise here, semantics and efficiency.

Semantics. A question arises about the effect of the cut. One reading of the cut in the first clause is that it removes all choice points made by the preceding goals in the clause instance. That is, (working backwards

through the choice points), the alternative solution to `c` is discarded, the alternative in the disjunction (i.e. the solution involving `d`) is discarded and the alternative solution to `b` is discarded, leaving the clause instance determinate up to that point. Thus, if the call `a,fail` was made, the result should be the single line:

```
b1c1e
```

However, if disjunction was implemented using the procedure above, the disjunction `(c,!;d)` would result in `call((c,!))`, where the effect of the cut would just be to discard the alternative solution to `c`, so for the same call `a,fail`, the result would have multiple solutions (separated here by commas):

```
b1c1e, d1e, d2e, c2e, d1e, d2e,
b2c1e, d1e, d2e, c2e, d1e, d2e
```

ISO Prolog supports the former interpretation, and a useful rule of thumb relates to the 'transparency' [54] of a construct: if the name of a construct contains alphanumeric characters, it is 'opaque' to a cut, i.e. the scope of a cut inside the construct is limited to within it. On the other hand, if the construct's name is exclusively non-alphanumeric, it is 'transparent' to a cut, and the scope of a cut inside the construct extends out into the clause instance that contains it. Thus, for example, disjunction, with the functor `;/2`, is transparent, whereas `fail_if/1` is not.

Efficiency. Even if the semantics of the cut were not an issue, implementing control constructs with built-in predicates as outlined above is likely to be computationally expensive, because of the processing typically necessary for the `call/1` predicate. In Open Prolog, for example, the argument to the `call/1` predicate must be turned into a procedure call

before it can be executed. In some compiler-based implementations, calls made from a `call/1` are interpreted, and compiled code has to be made public so that it can be called from the interpreter.

One way to implement control constructs efficiently is to transform the source code containing control constructs into equivalent 'canonical' construct-free code, preserving the original semantics. This approach is taken in many compilers, for example, in Prolog by Peter Van Roy in [70] and Andrew Taylor in [68], and by the implementors of Gödel [36], but it is not suitable here since the correspondence between the original source and any resulting image code would be difficult or impossible to identify later.

The approach taken in Open Prolog is conceptually straightforward—goals within control constructs are compiled normally, the control constructs themselves are compiled into instructions that bracket (i.e. precede and follow) the instructions for the goals they control. This approach eliminates the performance penalty that would be incurred if a variant of `call/1` was used. One other concept was introduced to help implement control constructs, the idea of *Non-Removable Frames*.

Non-Removable Frames, or *NR Frames*, are so called because they can not be automatically deallocated or discarded during normal execution, e.g. by last call optimisation. (They can be discarded on backtracking, however.) Furthermore, an NR frame can be made *transparent* or *opaque* to cuts, and this is signified by a flag bit in the frame.

7.3 Non-Removable Frames & Transparency

As stated, an NR frame can not be removed automatically in normal execution, either by last call optimisation or by the cut. This is accomplished at very low runtime cost by making an NR frame a choice point and by modifying the operation of the cut to honour the existence of NR frames. When an

NR frame is no longer needed, it can be made determinate (i.e. no longer a choice point), and is then available for deallocation by last call optimisation in the normal way. If it's at the top of the local stack, it can be deallocated straight away.

To enable the cut to handle NR frames efficiently, a new OPAM register, the `NR` register, is introduced. This register always points to the most recently allocated NR frame. By comparing the cut frame address with the `NR` register contents, the presence of NR frames above the cut frame can be established quickly.

If no NR frame is newer than the cut frame, the cut deallocates all frames newer than the cut frame, and the cut frame itself is made determinate—this is the cut's 'normal' behaviour. If the topmost NR frame is newer than the cut frame, (established by a comparison of two OPAM registers), then, as a preliminary step, stack frames newer than the NR frame are discarded. The NR frame itself remains a choice point, and whatever alternatives exist at that frame are unaltered.

If the frame is opaque to cuts, nothing further is done—no more choice points are discarded. If the frame is transparent to cuts, then all the choice points below the NR frame, but above the cut frame, (or another opaque NR frame if that comes first), are made determinate. Thus, the normal semantics of the cut are preserved, but it doesn't remove any NR frames. The non-removable property of NR frames is central to the implementation of the negation, *if-then*, *if-then-else* and catch constructs.

7.4 Disjunction

Consider the clause:

```
trial(X,Y) :-
        member(X,[a,b]),(jack(X);jill(Y)),write(Y).
```

This disjunction in this clause is encoded (in OPAM-like pseudocode) as:

```
        orCall(L1,L2)        %start of disjunction
        call jack(X)
        jump(L2)            %end of first disjunct
                            %and start of second
label(L1)
        call jill(Y)
        punctuation         %end of disjunction
label(L2)                   %continuation
        ...
```

The calls to `jack(X)` and `jill(Y)` are preceded, separated and terminated by the code for the disjunction, `orCall(L1,L2)`, `jump(L2)` and `punctuation` respectively. Assuming the call jack(X) will succeed, the instructions execute as follows:

`orCall(L1,L2)` A built-in predicate called `'system$disjunction'` is called. This predicate has two alternatives, so a normal choice point is constructed. The first alternative tags the choice point as a Disjunction Frame[1] and succeeds.

`call jack(X)` This is a normal call to the predicate `jack/1`. this goal is part of the body of the original clause, and executes in that clause instance's environment. The only effect of the disjunction has been to put a newer choice point on the stack. We assume `jack(X)` succeeds.

`jump(L2)` This instruction skips the code for the second alternative.

If backtracking should later cause the alternative of the disjunction to be called, the following will happen:

[1]This is needed as the OPAM distinguishes between the *if-then* and *if-then-else* at runtime by looking for a Disjunction Frame tag in the most recent choice point. See section 7.6.

`orCall(L1,L2)` Since this is the second and last call to `'system$disjunction'`, the choice point becomes determinate and the second procedure for `'system$disjunction'` is called, which simply jumps to the label `L1`, i.e. the start of the second alternative.

`call jill(Y)` Again, this is a regular call, being made from the body of the original clause, and may succeed or fail. Should it fail, backtracking occurs as normal, since the choice point placed on the stack by the `orCall` is no longer a choice point. If the call to `jill(Y)` is successful, the next instruction is executed:

`punctuation` This is a no-op instruction, used in the reconstruction of the source code; it marks the end of the disjunction.

Disjunction and the Cut

Whether a cut is placed inside the disjunction or elsewhere in the clause body, it will be compiled as part of the clause body containing the disjunction, and will cut choice points back to the same frame, i.e. the frame of the clause instance of which it is a member. Thus, the presence or absence of a disjunction makes absolutely no difference to the cut's operation. In this sense, the disjunction is transparent to the effects of the cut, and no special handling has proven necessary.[2]

7.5 Negation

Open Prolog implements four variants of negation. Two are transparent to cuts, viz. `\+ /1` and ¬`/1`; two are opaque, viz. `not/1` and `fail_if/1`. Each is encoded as a variant of the `notCall`. Consider the clause:

[2]If it was necessary to implement an 'opaque' disjunction, the frame placed on the stack by the orCall would have to be made into an opaque NR frame.

```
test(X) :-
        fail_if(jack(X)).
```

This is encoded as:

```
        notCall(3,L1)       %start of negation
         %(fail_if is variant 3)
        call jack(X)
        notSucceed          %end of negation
label(L1)                   %continuation
```

The call to `jack(X)` is preceded and terminated by the code for the negation, `notCall(3,L1)` and `notSucceed` respectively. The general plan of the implementation is somewhat similar to that of disjunction, and is described in the following paragraphs.

`notCall(3,L1)` A built-in predicate called `'system$not'` is called. This predicate has two alternatives, so a normal choice point is constructed. However, the first alternative turns the choice point into an NR frame by storing the current value of the NR register in the NR_field of the frame, and updating the NR register to point to itself. Next, since this is an opaque variant of negation, the frame's transparency bit is turned off and the predicate succeeds.

`call jack(X)` This is a regular call, being made from the body of the original clause, and will succeed or fail. If it succeeds, the next instruction will be `notSucceed`; if it fails, program execution will backtrack to the second `'system$not'` predicate. Now, this being negation-as-failure, if `jack(X)` succeeds, `fail_if(jack(X))` must fail; `if jack(X)` fails, `fail_if(jack(X))` must succeed.

If `jack(X)` succeeds, the instruction `notSucceed` is executed. This instruction makes `jack(X)` determinate, resets the `NR` register to the value of the NR_field of the NR frame, thus removing the frame, from

the NR chain and effectively removing its special status as an NR frame. The instruction concludes by executing a failure; thus, the success of the predicate `jack(X)` causes failure.

If `jack(X)` fails, program execution backtracks to the second `'system$not'` procedure. This procedure also resets the `NR` register, relinquishing the frame's NR frame status. OPAM program execution is transferred to the instruction following the negation, labelled `L1`; thus, the failure of the predicate `jack(X)` causes success, and program execution continues at the instruction following the negation.

7.5.1 Negation and the Cut

Consider the clause:

```
test(X) :-
        fail_if((!,jack(X))).
```

The cut is placed inside the negation will be compiled as part of the clause body containing the negation. Now, if the frame placed on the stack for a negation was not an NR frame, execution of the cut would incorrectly make its clause body determinate, as if the negation was transparent. Even, however, if the variant of negation used *was* transparent, the cut would also remove the negation's own choice point, and if `jack(X)` were then to fail, (meaning that `fail_if(jack(X))` should succeed), the second `'system$not'` procedure would not be called, and the negation would thus, incorrectly, fail.

7.6 *If-Then* and *If-Then-Else*

The control constructs *if-then* and *if-then-else*, widely used in Prolog, and standardised in ISO Prolog, are closely related.

The *if-then* construct is the infix functor `->`. Its first argument may be thought of as a *test* predicate, and its second argument an *action* predicate. Informally, the semantics of *if-then* are that if the test predicate succeeds, its alternatives are discarded and the action predicate is executed. Thus, for example, the informal meaning of the term `try(T)->do(X)` is that if `try(T)` is true, discard its alternatives and call the goal `do(X)`. Backtracking will not attempt to re-prove `try(T)`, as its alternatives will have been discarded.

The *if-then-else* construct is a combination of the *if-then* functor with a disjunction. The construct takes three arguments that can be thought of as a test predicate, an action predicate and an *alternative action* predicate. Informally, the semantics of an *if-then-else* construct are that if the test predicate succeeds, its alternatives are discarded and the action predicate is called; otherwise the alternative action is called. For example, the informal meaning of the construct `try(T)->do(X);do(Y)` is that if `try(X)` is true, discard its alternatives, discard the disjunction's alternative (i.e. discard the possibility of proving the disjunction later by executing its second alternative) and call the goal `do(X)`, else call the goal `do(Y)`.

Structurally—that is, looking at these constructs as structured Prolog terms—*if-then-else* is a disjunction in which the first alternative is an *if-then* term. For example, the *if-then-else* term `try(T)->do(X);do(Y)` can be rewritten to emphasise its structure thus: `(try(T)->do(X));do(Y)`, where the first alternative is the *if-then* term `try(T)->do(X)`.

Semantically, *if-then-else* is more than just the combination of the independent semantics of a disjunction with a subsidiary *if-then* because an *if-then* within a disjunction makes the disjunction determinate.

7.6.1 Implementation

Implementation of *if-then* and *if-then-else* follows the format of the other control constructs. If the construct is *if-then-else*, then the disjunction is compiled as described in Section 7.4. To encode the *if-then* construct, the test and action predicates are preceded, separated and terminated by the OPAM instructions `ifThenCall`, `ifThenCommit` and `punctuation`. As an example of *if-then*, consider the clause:

```
trial(T,X) :-
        try(T)->do(X).
```

The *if-then* is encoded as follows:

```
        ifThenCall
        call try(T)
        ifThenCommit
        call do(X)
        punctuation
```

Assuming `try(T)` succeeds, the instructions operate as follows:

`ifThenCall` A built-in predicate called `'system$if$then'` is called. This predicate has two alternatives, so a choice point is allocated. The first alternative converts the choice point into an NR frame. (The frame is used only for the test predicate, and is then discarded.) The *if-then* construct is transparent to cuts, so the frame's transparency bit is turned on.

`call try(T)` This is a regular call, being made from the body of the original clause. If it succeeds, the next instruction will be `ifThenCommit`; if it fails, program execution will backtrack to the second `'system$if$then'` predicate.

`ifThenCommit` First, the test predicate is made determinate by removing the NR field constructed by the `ifThenCall`. Then, the behaviour of the instruction depends on whether the construct is an *if-then* or an *if-then-else*. The most recent choice point is examined, and if it is the choice point of a disjunction, then this code is taken to be part of an *if-then-else* construct; in that case, the disjunction is made determinate and the instruction succeeds. If the most recent choice point is not a disjunction, the instruction does nothing further and succeeds.

`call do(X)` This is a regular call, being made from the body of the original clause, and will succeed or fail. Since the outcome of the test predicate `try(T)` has been made determinate, the test can never be re-satisfied on backtracking.

`punctuation` This is a no-op, used to assist in the reconstruction of the source code.

If the test predicate (`try(T)` in this example) fails, then backtracking causes the second alternative `'system$if$then'` to be called. This instruction removes the NR field placed on the local stack by the `ifThenCall` instruction and fails. If it's an *if-then-else* construct, then this failure will backtrack to the second alternative within the disjunction as required.

7.7 Catch-and-Throw Exception Handling

Catch-and-Throw exception handling is a familiar technique in conventional procedural languages and was adopted as part of the ISO Prolog standard. The technique is based on the metaphor of a message or signal being 'thrown' from where an exception occurs back to a line of 'catchers', arranged with the newest catcher first and the oldest catcher last. If the first (i.e. the newest) catcher can deal with the error, then it retains the message and processes

it. If not, the message continues on its 'flight path' back to the next catcher which repeats the process, and so on down to the last (and oldest) catcher—typically part of the runtime system—which aborts the offending program. Overall, the idea is that the most local catcher that can handle the exception gets the first opportunity to deal with it.

In Prolog, the message is a Prolog term, and catchers are `catch/3` predicates. The arguments of the `catch/3` predicate are as follows:

- The first argument is the goal—the *subject*—that is executed, and exceptions thrown by this goal or its subgoals will be processed by this `catch/3`, unless newer `catch/3` instructions within the goal or subgoal catch the exceptions first.

- The second argument is a term with which the message term that has been thrown must unify for this `catch/3` predicate to be able to deal with the exception in question.

  When an exception occurs during the execution of the subject or one of its subgoals, a message term is thrown and may make its way back to this catcher. The message term is unified with this, the catcher's second argument; if unification is successful, the third argument is called as a goal. If the message term fails to unify with this argument, then the message is passed on to the next oldest `catch/3` predicate, and this catcher is effectively bypassed.

- The final argument is a goal—the *exception handler*—that is called if the message term thrown by the exception unifies with the second argument.

7.7.1 Implementation

The `catch/3` predicate is implemented similarly to the control constructs. It contains two goal calls, the subject and exception handler, and the second argument is a term. These three items are compiled as part of the body code of the clause containing the `catch/3` call. To ensure that the second argument is compiled as a term rather than a goal, and to force any variables it contains to be globals, it is made the subterm of a structure (`skip/1`) which becomes the only argument of a call to a predicate (also `skip/1`).[3] For example, consider the following clause:

```
trial(X,Y,Z) :-
        catch(do(X),catch(Y),handler(Z)).
```

The catch predicate is compiled into the following OPAM code:

```
        catchCall
        call do(X)
        catchSucceed
        jump L1
        call skip(skip(catch(Y))) % force catch(Y) to be a term
        cut
        call handler(Z)
        punctuation
label(L1)
```

When an exception throws a term, the term must be tested for unification against each catcher's second argument in newest-to-oldest order until a successful unification is performed. To do this, the catcher's environment, (or part of it), must be restored so that the unification can be attempted. This is done by locating the local frame of the catcher, restoring from it the local and global stack pointers that were current for that clause instance. To implement this efficiently, the OPAM uses a chain of special local frames called

[3]The structure and the predicate chosen have no function or significance beyond forcing the compiler to compile the second argument as a term containing no local variables.

Catch Chain Frames each of which points to a particular catcher's frame and also to the next oldest member of the catch chain frame. In addition, an OPAM register called the `CCR`, the *Catch Chain Register*, always points to the newest catch chain frame. Thus, the unification environment for each catcher can be located rapidly when an exception occurs.

The instructions operate as follows:

`catchCall` The local stack frame of this instruction's clause instance becomes the newest catch frame (i.e. the local frame of the catcher's environment); the purpose of the instruction is to update the catch chain with this information. To accomplish this, the instruction calls a built-in predicate named `'system$catch$predicate'`, which has two alternatives, so creating a new choice point is placed on the local stack. This new choice point becomes the latest element of the catch chain and is linked into the catch chain, as follows:

- The current value of the `CCR` is stored in the frame,
- The `CCR` is made point to this frame,
- The frame's Continuation Frame pointer (the `Xp/CF` field of the frame) already points to the catch frame it refers to,
- To ensure the correct semantics of the cut, the new frame is turned into an opaque NR frame.

`call do(X)` This is a normal call to `do(X)`.

`catchSucceed` At this point, the subject of the catch has succeeded, so the `catch/3` instruction must be removed from the top of the catch chain and the previous catch frame restored to the top of the catch chain.

If the subject of the `catch/3` is determinate, the catch chain frame constructed by the `catchCall` instruction is no longer needed and is discarded. It is removed from the catch chain and the NR chain, restoring

both the CCR and the NR registers to the values they held prior to execution of the catch/3 predicate. If the subject of the catch/3 is non-determinate, however, the possibility exists that program execution could backtrack to the subject in the future, and the catch chain would then have to be restored to its present state. In the non-determinate case, therefore, the catch chain frame created by the catchCall instruction is removed from the NR chain but not deallocated. This also restores NR to its prior value, before the catch/3 predicate was executed. A built-in predicate 'system$catch$succeed' creates a new choice point. The current value of the CCR is stored in it, and the previous value of the CCR is restored from the catch chain frame created by the catchCall. The new local frame now contains sufficient information to reconstruct the catch chain and the NR chain if backtracking returns program execution to the non-determinate subject of the catch/3 predicate.

The catch-and-throw mechanism as described does not comply with the semantics prescribed in ISO Prolog. Consider the following sequence:

```
\verb"catch((X=4,throw(X)),J,write(J)),write(','),write(X)".
```

The ISO Prolog standard calls for the following result: 4,Q, where Q is an unbound variable. Open Prolog will, however, respond with 4,4; that is, X remains bound to 4 after the catch has been made. Operationally, the difference between the two is that when the 'ball' has been thrown and successfully 'caught', all variable bindings made since the first argument was called are undone to conform to the ISO requirement. So, while X was unified with 4 while the X was thrown, as soon as it was received, X was uninstantiated. Crucially, Open Prolog does not undo the bindings of variables made since the first argument was called.

7.7.2 Related Work

The scheme adopted here of `NR` frames and bracketed code sequences is similar to, (but developed separately from), the scheme used to implement disjunction in MProlog [43, Section 9, p8]. In MProlog. In MProlog, alternatives in a disjunction form 'groups' that are treated as parameterless procedures which create a stack frame to store the choice point needed. In Open Prolog, alternatives in a disjunction are compiled in the normal way but are then surrounded by special instructions, effectively forming them into groups also. The special instructions have the effect of generating and managing stack frames, just as the parameterless procedures do in MProlog. In Open Prolog as in MProlog, special care is taken of the interaction between the cut and the disjunction. The scheme is more generally applied in Open Prolog however, being used, with modifications, to handle negation, if-then, if-then-else and, trivially, nested conjunction. It seems that disjunctions of more than two alternatives are contemplated in MProlog, whereas only two are allowed in Open Prolog.

Another approach to implementing if-the, if-then-else and negation is described by Bowen *et al* in [8]. Cut is compiled to instructions that take an explicit argument defining the scope of the cut, obviating the need for `NR` frames or the like in such circumstances.

Chapter 8

Memory Management

8.1 Summary and Contributions

This chapter describes how memory usage is handled in Open Prolog. Memory is reallocated dynamically to four sections: code space, global stack, local stack and trail. Management of the adjustment of the sizes of the spaces is discussed, garbage collection of the global stack is described and garbage collection of the code space is discussed.

The contributions are: the implementation of garbage collection in a structure sharing implementation using the logical update view of dynamic code, and a simple scheme for the avoidance of some of the runtime overhead due to the logical update view. Reference chains are 'rolled up' where possible.

8.2 Memory Management

The layout of OPAM memory is shown in Figure 4.8 on page 97. The Name Space is allocated a fixed amount of memory at startup[1] and is not managed at runtime. In particular, the amount of memory is not changed at runtime, and functors and atoms added to the name space at runtime are never removed; thus the possibility of name space overflow exists. All other parts of OPAM memory are actively managed at runtime: the global and locals stacks and trail are dynamically adjusted—resized and remapped, the global stack is garbage collected and retracted clause space is garbage collected from the code space.

Triggering Memory Management Activity

Each space in OPAM Memory has three parameters: pointers to the lower and upper bounds of the space and a pointer to the upper operating limit. During a `neck` instruction, a check is made to see if the operating limits are being exceeded in the stacks or in the trail. Similarly, when a clause is about to be asserted into the Code Space, a check is made to ensure that the upper operating limit of the stack space would not be exceeded by the addition of the extra code. If necessary, a 'stack adjustment' is started which will resize and remap these spaces. If insufficient space is available to make the process worthwhile, a garbage collection will be attempted first. In Open Prolog, two different kinds of garbage collection are done—one on the global stack, the other on the code space, recovering space from clauses that have been retracted and are no longer accessible from a running program.

[1] Memory allocation parameters, including initial allocations and memory management parameters, are set in `STR#` resource 130 in the application.

8.3 Stack Adjustment

Stack adjustment is the first and computationally the cheapest form of memory management, and involves the code space, the global stack, the local stack and the trail. These occupy a contiguous block of memory at the upper end of OPAM memory and inside that block there are three boundaries—between the code space and the global stack, between global stack and the local stack and between the local stack and the trail. Stack adjustment involves recalculating the boundary points, safely moving the global and local stacks and the trail to their new locations and finally remapping pointers to elements of the local stack and the trail. Code in the code space is not moved during stack adjustment.

8.3.1 Implementation

Recall that a stack adjustment can be sought for two reasons: if one of the stacks exceeds its operating limit or if a clause is about to be asserted. In the latter case, a specific extra amount of code space will be sought.

- If the amount of free space at the top of the code space would be less than the minimum threshold, an allocation of code space equal to the threshold plus the size of the code about to be added, if any, is sought.
- The total amount of space used by the three stacks is calculated.
- The total amount of space that would be available to the three stacks is calculated, allowing for the demand for extra code space.
- If the total amount of free space available relative to the amount of space used by the three stacks falls below a set threshold,[2] a garbage

[2]This is set by item 3 in STR# resource 130; at present, it is 200 bytes free space for every kilobyte in use.

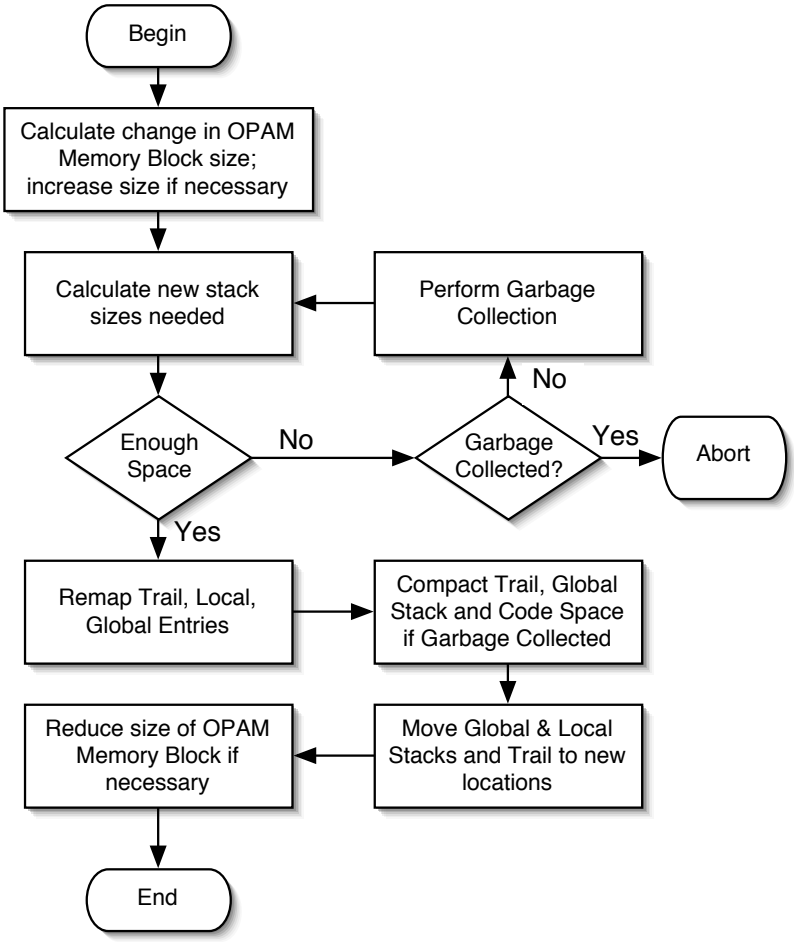

Figure 8.1: Stack Adjustment and Memory Management. Each time a `neck%` instruction is executed, a check of the stacks and trail is made. If there is insufficient free space, a stack adjustment, outlined here, is started.

collection is performed. Afterwards, if the free space is still below the threshold, the program is considered to have run out of memory and is aborted.

- The new allocations for the stack spaces and the code space are calculated: the code space is resized exactly as requested, and the remaining space divided among the three stacks according to their current usage.

  A stack adjustment may move the base of all three stacks, though the global stack will normally only be moved if code is about to be added or has been garbage collected from the code space; the base address of code space itself is never affected.

- The local stack is traversed from newest frame to oldest, and every pointer associated with each frame is remapped. With a simple stack adjustment, remapping simply has to allow for the new base address of each stack, and no code space remapping is necessary; however, if a garbage collection has been performed, remapping must also allow for the amount of garbage recovered below each pointer in its stack or space. The calculations necessary to determine these amounts during remapping are performed during garbage collection.

8.4 Garbage Collection

One of the delights of Prolog is that the programmer doesn't have to be concerned with the minutiae of data structures; in many cases, the programmer can remain unaware of the management issues and processes. One of those issues is garbage collection, the recovery of areas of memory that have been brought into use, but which are no longer needed and have to be released back to the system for reuse. For a thorough survey of garbage collection techniques, the reader is referred to [41]. In Open Prolog, garbage collection

is done on the Global Stack and on the Code Space. The process is described in this section.

From the previous section, recall that garbage collection is initiated during a stack adjustment when the combined amount of free space available in the Code Space, Global Heap, Local Heap and Trail is less than a set threshold.

Garbage collection activity always starts at a `neck` instruction or just prior to asserting a clause, so the state of execution of a program is captured in the state of the stacks and the OPAM registers. The local stack contains the frames of all goals that are still active, in addition, possibly, to some inactive frames. A frame is active if it can be reached by normal execution or by backtracking. Thus, the chain of forward execution that can be followed by dereferencing the `Xp/CFP` fields of successive local frames, and the chain of 'backward' execution that similarly can be followed via the `VV` fields of local frames must both be followed to locate all reachable items in memory. (The Non-Removable Frame Chain and the Catch Chain do not need to be followed, as the frames in these chains are guaranteed to be in the forward or backward execution chain.)

The global stack contains frames of global variables associated with active local frames, and in addition may contain frames of global variables whose local frames have been deallocated because the goal completed determinately.

Garbage collection is done in the following steps:

Variable Cell Marking A reachability analysis marks every variable cell accessible to the program. In a scheme due to Warren in [72], marking is done in two phases to minimise the risk of deep recursion while marking constructed terms.

Code Marking If there has been database activity (i.e. the assertion or retraction of clauses), a reachability analysis is also performed on public clauses in the code space to identify clauses that have been retracted

but that are still accessible to the program.

Remapping Calculations Preliminary calculations are made to enable pointers to be remapped. For instance, the offset by which each global frame will be moved—and hence the offset to be added to every pointer to variable cells within the frame—is calculated.

Remapping The local stack is traversed from newest frame to oldest, and every pointer associated with each frame is remapped, making allowance for the removal of garbage and the moving of the stack itself.

Compaction The stacks are compacted by copying all marked items to the low end of the stack space. Similarly, the code space is compacted by moving all accessible clause code to the low end of the code space.

Moving Having been compacted, the stacks are now safely moved to their new locations.

Reset Call Action Procedures The call action procedures used to access clauses are reset.

As a preliminary to the marking phase, a *Clause Retraction List*—an ordered list of the start and end addresses of every clause that has been retracted—is generated. Even though a clause might be retracted, it might not be possible to actually remove a clause's code, as it may still be in use. Thus, the list represents all *potentially* removable clauses.

8.4.1 Marking

Marking is done in two phases to reduce the amount of recursion likely to be needed to trace reachability through deeply nested structures.

Phase One

In the first phase, variables in active frames are marked. Starting with the frame at the top of the local stack, the local variables in the frame are marked; space for them lies between the `Vp_CutField` and the memory location the `Vp_CutField` points to. This space may hold local variable cells or it may hold 'reserved' local stack space; if so, the reservation is preceded by a cell containing the *gcReserved* tag and the length of reservation in bytes. Reserved space is not marked.

The local frame's global frame is reached via the frame's `V1_Field` and the global frame of the continuation is reached via the frame's `X1_Field`. The number of variables in a global frame is not known in advance; instead, a cell marked with the code `gcMark` is always placed at the start of each global frame, and the end of the frame is found by encountering the `gcMark` cell of the next frame.

Clause Code Marking

Also in the first phase, clause code used in active clause instances is marked. If clauses have been asserted or retracted since the clause instance was instantiated—easily determined by comparing the frame's `IT` field with the current `DBC` register—the clause code used by the clause instance is marked. To locate the start of the clause code, the body of the clause is 'walked' to the end, where the size of the clause code is to be found.[3] This is subtracted from its own location to give the location of the start of the clause's clause descriptor record, where the mark is recorded. If the local frame is a choice point, then each remaining alternative clause is also marked. When the code space is finally compacted, only retracted clauses that have not been marked will actually be removed.

[3]This is similar to a technique used (but unpublished) in Quintus Prolog: compiled clause code ends with an 'end-of-clause(size)' marker [20].

When a local frame has been processed, the next local frame is located using the greater of its continuation frame pointer and the current backtrack pointer.

Phase Two

In the second marking phase, variables referenced through constructed terms are marked. Iterating down the local stack, every frame variable, though marked in the first phase, is dereferenced; if it dereferences to a variable that's already marked, nothing further is done, because either the dereferent is marked because it is part of a frame, and will therefore be processed in its own right as a frame variable, or it has already been marked during the reachability analysis. Otherwise, the dereferent is marked, and, if a constructed term, it is recursively analysed for reachability.

As an optimisation, dereference chains are 'rolled up' during garbage collection. A dereference chain is a maximal length chain of unmarked reference cells, ending with a non-variable or with an uninitialised variable, which is represented as a cell that references itself. Each dereference chain is replaced by a single link when remapping is being done, so intermediate variables on a dereference chain are not marked. A similar technique for the WAM, called *Variable Shunting* is described by Sahlin and Carlsson in [59]. Apart from shortening the dereference chain, and possibly speeding up later access to it, the removal of intermediate references in a dereference chain has the specific benefit in a structure sharing implementation of improving the likelihood of being able to remove the global frame in which the intermediate reference is held, so long as the global frame is inactive.

The Trail Stack is then traversed, marking entries that point to unmarked items (i.e. garbage). These are entries that will be removed from the Trail Stack during the compaction phase.

Clause Code Marking

Also during the second marking phase, the clause containing the skeleton code of each constructed term is marked if there have been assertions or retractions since the last garbage collection. The marking process is slightly different to that described for phase one. At the start of garbage collection, a list of the starting and ending addresses of every clause that has been retracted is made. During phase two, each time a constructed term is marked, a binary search is made of the list, to see if the skeleton's address lies within a clause that has been retracted; if so, the skeleton is part of the clause, which is therefore still reachable and is marked for retention.

While Warren's scheme for reducing the depth of recursion during the marking phase certainly works, it relies on deeply nested structures being associated with active frames. Unfortunately, aggressive memory management techniques have the side effect of reducing the number of active frames, reducing the likely effectiveness of the technique. Accordingly, a separate stack, which is allocated space in a relocatable block on the heap in the application zone, is used to accommodate the very deep recursion sometimes encountered during the marking phase.

8.4.2 Remapping Calculations

At this point, all reachable local stack items, global stack items and clauses marked for retraction but still in use have been marked. Garbage trail entries have also been located and marked. The space currently occupied by 'garbage' can thus be readily identified. This space will be recovered by moving—*'compacting'*—non-garbage items to one end of the stack they occupy. In Open Prolog, compacting is done on the code space, the global stack and the trail. No attempt is made to recover space from the local stack.

Before the compacting phase, however, it is necessary to adjust pointer

values to the new locations of their targets. To do this, it is necessary to calculate the offset of each pointer or group of pointers.

Trail

The total number of bytes by which the trail will be compacted is calculated—this is the amount by which the stack will contract due to the removal of garbage trail entries.[4] This is also the amount by which the very last trail entry will move when the stack is finally swept, so it represents the amount by which a pointer to the last trail entry must be offset when it is updated. Thus, the `TR` field in the topmost frame will be adjusted by the trail compaction less the space occupied by any garbage trail entries in the frame's list of trail entries. This is found by traversing the trail from the top (i.e. the newest end) to the entry pointed to by the frame's `TR` field. This calculation also yields the amount by which the rest of the stack will be compacted, and this can be applied to the next frame in the same way as the overall compaction was applied to the first one. At all times, therefore, it a 'running' compaction figure is maintained, frame by frame, that is easily adjusted to become the offset to be applied to each frame's `TR` field.

Global Stack

Every global frame has a one-cell sized reservation at the start of it for use during garbage collection called the `GC Cell`. It contains a tag and a 4-byte data field. The tag is normally `gcMark`, and the data field is initially unused, though some built-in predicates put a four character signature there to assist low-level debugging. During this phase of the garbage collection, the global stack is traversed from bottom to top, identifying the offset by which each global frame will be displaced when the stack is compacted and storing it

[4]These are entries in the trail stack that point to variables that have not been marked and are thus garbage.

in the data field of the `GC Cell` of each frame. If a frame is all garbage, its `GC Cell`'s tag is set to `gcDiscard`, indicating that this entire frame will be discarded completely during compaction. (A frame is considered all garbage if there are no marked cells between it and the `GC Cell` of the next frame.)

Code Space

The entries in the name table for clauses in the code space must be adjusted to point to the clauses at their new locations.

First, a traversal of the public code space is made and the amount by which each clause will be moved during the compacting phase is calculated by adding up the amount of space to be recovered below each clause in the code space. This offset, calculated for each clause, is stored in the `delta` field of the clause's descriptor record.

Next, a traversal of the name table is done, looking for functors with associated clauses. The pointer to the first clause that is stored in the functor entry is adjusted by the clause's `delta` value.

8.4.3 Remapping

Once again, every local frame is traversed, starting at the newest and working backward through the local stack. For every local frame, every pointer in it, and every pointer that can be reached from it is remapped.

Trail Pointers

Each choice point has a valid `TR` field which must be adjusted by the number of bytes of garbage that will be removed from below it in the trail stack, as well as by the offset to the new base address of the trail. The garbage count starts with the total number of bytes to be removed from the trail. This is updated as each choice point frame is processed by traversing the trail entries

for the choice point, reducing the garbage count for each garbage trail entry found.

Local Frame Pointers

No garbage is removed from the local stack, so remapping local stack pointers only involves adjusting for the new location of the local stack base.

Global Frame Pointers

References are dereferenced to the first marked item in the chain and replaced by a reference to that dereferent. Of course, the dereferent may itself be a reference or a constructed term, in which case it must be remapped.

As with remapping of other pointers, the remapping offset is the sum of the stack removal offset and the garbage offset. The garbage offset of a global pointer is the garbage offset of its global frame, which has already been calculated and stored in a GC Cell just below the first active cell in the frame. Thus, to find the garbage offset of a global reference, it is necessary to iterate down the frame containing the reference until the `GC Cell` containing the frame's garbage offset is found.

Molecule Frame Pointers

In the current version of Open Prolog, all constructed terms are global, and this means that the frame of variables referenced in a constructed term must be on the global stack. Since these variables are referenced using the frame pointer as a base address, frame pointers must be remapped in the same way as other global stack pointers.

It is not safe to assume that a frame pointer still points to an active global frame, as the clause instance that gave rise to the global frame may have completed determinately. This being so, the bottom of global frame

may have been trimmed in a previous garbage collection and the frame's GC Cell, containing its garbage offset, would have been moved upwards towards the remaining variable cells in the frame. Consequently, a constructed term's frame pointer could be pointing into a different global frame, and the garbage offset for that frame would be inappropriate.

To be certain that the correct global frame is accessed, the skeleton of the constructed term is 'walked' until the first variable reference, if any, is found. This reference gives the offset from the base of the global frame to the variable, and so the corresponding variable cell can be accessed. By iterating downwards from *this* cell, the garbage offset of the cell's global frame, and hence of the constructed term's global frame is located.

Code Space Pointers

Pointers to code space need to be adjusted by the delta value stored in their Clause Descriptor Record. If a code space pointer is below the address of the first retracted clause, its location is unchanged. If it is above the highest retracted clause, its new location is offset by the total saving of code space. If it is between the first and last clause, a binary search of the Clause Retraction List is performed to determine the delta value to be added.

Chapter 9

Evaluation

In this chapter, the performance of Open Prolog is firstly compared with other implementations on the same machine, using widely-available benchmarks. Next, the profiling mechanism built in to Open Prolog is described. Finally, profiling results for some standard benchmarks are presented and discussed.

Background

The first Prolog implementation available on the Macintosh was *MacProlog*, written by Frank McCabe originally for the Apple Lisa computer [21]. This was available in a free version and a considerably faster commercial version. In the mid-90s, according to Brian D. Steel [64], the inference engine was replaced by one derived from Steel's 386-PROLOG design and renamed MacProlog32. This is still available from Logic Programming Associates [81], though it is not a commercial product anymore. It still features two modes of operation, normal and 'optimised'.

Open Prolog was first posted to the internet in the middle of 1991. It ran on Macintoshes with 68000 processors or better, having a minimum of 1 MB of physical memory—i.e. any Macintoshes from a Mac Plus onwards. An implementation of *Tricia* Prolog [87] was ported to the Macintosh by a team

at Uppsala University,[1] based on work by Mats Carlsson and others [17, 4, 3] in the years 1989–1993. Later on, but before 1995 [19], *SICStus Prolog* [18, 2] was ported to the Macintosh 68020. Subsequently, the Macintosh moved from the Motorola 68000 family of processors to the Motorola/IBM PowerPC family. An emulator for the Motorola 68LC020 instruction set provided as part of the Mac OS allowed programs written for the older processor family to run on the new one.

For a considerable number of years, therefore, there were two free Prologs—Tricia and Open Prolog—and two licensed/commercial Prologs—MacProlog (or MacProlog32) and SICStus—available on the Macintosh, all running on the 68LC020 emulator. During this time, Open Prolog was widely used.

To the author's knowledge, no other Prolog implementations were released for the Macintosh until the more recent transition of the Macintosh operating system itself to a platform based on FreeBSD. This enabled the porting of popular UNIX implementations of Prolog, notably GNU Prolog [30], SWI Prolog [86] and SICStus Prolog. Thus it is now possible to run UNIX implementations of Prolog alongside earlier Prolog implementations in the Classic Mac OS environment on the same machine.

Performance Evaluation

The performance of Open Prolog was evaluated in terms of its speed relative to other implementations. In addition, performance bottlenecks were identified by running programs on a version of Open Prolog that was instrumented with profiling code. This code recorded usage and timing data for different sections of the application and the data was used to identify the most time-consuming parts of a program's operation. Profiling and profiling results are discussed in section 9.3 below.

[1] The developers and contributors to Tricia are listed in a splash screen reproduced on page 261.

9.1 Benchmarks

Tests were performed on a Macintosh PowerBook. The tests were suggested by results quoted on the performance summary page [85] on SICStus Prolog's web site. The benchmarks referenced on that page were downloaded and used with slight modifications. The benchmarks comprise the classic naive reverse benchmark and eight further benchmarks—*crypt*, *deriv*, *poly*, *primes*, *qsort*, *queens*, *query* and *tak*—referred to here as the *harness benchmarks*, as they are run within a 'harness' of test code.

9.1.1 The Naive Reverse Benchmarks

The Naive Reverse benchmark is the 'classic Prolog benchmark' and measures logical inferences per second while executing the *naive reverse* procedure `nrev/2`. Here is the code of the benchmark itself:[2]

```
nrev([],[]).
nrev([X|Rest],Ans) :- nrev(Rest,L), append(L,[X],Ans).

append([],L,L).
append([X|L1],L2,[X|L3]) :- append(L1,L2,L3).
```

The benchmark measurement is the time it takes to execute the call `nrev(X,_)`, where `X` is a list of numbers from 1 to 30. Since 496 calls are made, the number of logical inferences per second (LIPS) can readily be calculated.

The *nrev* and *nrev2* Benchmarks

Due to the low resolution of the system clock (16.625 milliseconds for early versions of the Mac OS and 10 milliseconds for UNIX), the benchmark must be iterated to build up sufficient execution time to enable a reasonably accurate average execution time to be calculated.

[2]The full benchmark code is listed on page 239.

In the *nrev* benchmark, an attempt is made to compensate for the overhead of iterating the call to the benchmark each time by performing the same number of iterations to call a dummy predicate. The time expended in doing so is deducted from the total to estimate the actual iterated benchmark time. This benchmark was used to generate the *nrev* benchmark data.

To check the accuracy of the figures generated by *nrev*, a more direct timing test was performed by constructing a clause with many sequential calls to `nrev/2`, and by measuring the time taken to execute the entire sequence of these calls. The clause:

```
timeTrial(X) :-
    data(L),
    statistics(runtime,_),
    nrev(L,_),nrev(L,_),...,nrev(L,_),
    statistics(runtime,[_,X]).
```

was generated, where the `nrev(L,_)` call is repeated 1000 times, using the `prepare/3` predicate on page 243, and this was used to generate the results in columns *nrev2*.

The `dynamic/1` Directive In most Prolog implementations, it is assumed that, once compiled or consulted, procedure code will not be altered by asserting or retracting clauses—procedures are assumed to be *static*. For a procedure to be alterable at run time, it must be declared to be *dynamic*.[3] To examine the effect of declaring a procedure dynamic, the *nrev* and *nrev2* benchmarks were repeated (as *nrev(D)* and *nrev2(D)* respectively) with the relevant predicates declared dynamic. (Open Prolog does not [need to] distinguish between static and dynamic clauses.)

	Performance in kLips			
Implementation	nrev	nrev(D)	nrev2	nrev2(D)
Open Prolog	806	806	758	758
Open Prolog, no GC	751	751	587	587
Tricia (C)	230	15	—	—
MacProlog32 (Opt)	1984	214	—	—
MacProlog32	213	213	—	—
SICStus 2 (C)	1462	162	1417	150
SICStus 2 (I)	158	158	153	151
GNU Prolog	1002	n/a	978	n/a
SWI Prolog	755	515	634	459
SICStus 3 (C)	4429	480	4299	454
SICStus 3 (I)	477	480	451	444

Table 9.1: 'Naive Reverse' benchmark results for Prolog implementations running on a 550 MHz Macintosh PowerBook G4. Results are kLips—thousands of logical inferences per second. Implementations above the double horizontal line are 'classic' implementations written in Motorola 68000 or 68020 code; these are executed on an MC68LC020 emulator built into the Mac OS. The other implementations run on the computer's PowerPC processor.

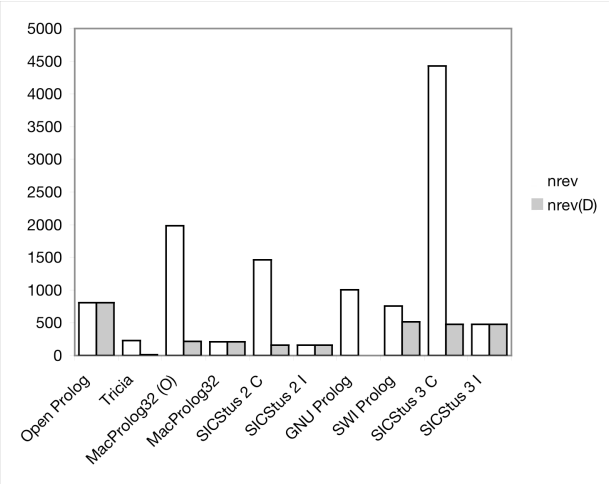

Figure 9.1: Naive Reverse Performance Chart. The performance of different Prolog implementations is charted for static code (clear bars, *nrev*) and for dynamic code (grey bars, *nrev(D)*). Open Prolog is faster than any other Prolog implementation in interpreted mode while offering the same ability to assert, inspect, debug and retract code. Likewise, Open Prolog is faster than other implementations where they are dealing with dynamic code. In terms of outright speed, however, and where these abilities are not needed, Open Prolog is considerably slower than SICStus 3 and 2 (compiled) and MacProlog32 (optimised).

9.1.2 *nrev* and *nrev2* benchmark results

Table 9.1 lists the performance of various Prolog implementations on a Macintosh PowerBook, and this is summarised in Figure 9.1. The four columns list performance in the standard *nrev* benchmark (column "nrev"), the same benchmark with `append/3` and `nrev/2` declared dynamic (column "nrev(D)"), the more direct *nrev2* benchmark (column "nrev2") and the same benchmark with `append/3` and `nrev/2` declared dynamic (column "nrev2(D)").

Four of the implementations tested—Tricia, MacProlog32, SICStus 2 and Open Prolog—were written for the Motorola 68000 or 68020, so they run on the MC68LC020 emulator provided as part of the Mac OS, and they also run within the 'Classic' compatibility environment of Mac OS X. Results for

[3]GNU Prolog doesn't permit dynamic predicates—all predicates must be static.

these 'classic' implementations are presented above the double horizontal line; results for implementations running directly on the host processor appear below it.

The annotation (C) means that the code was compiled; (I) means it was interpreted. MacProlog32 has a code optimiser—the annotation (Opt) means the optimiser was used on this benchmark.

MacProlog32 was version 1.25 running in 70,000 KB of RAM; Tricia was version 0.9.5a1 running in 4096 KB; Open Prolog was version 1.1b15 running in 4096 KB and SICStus 2 was version 2.1#9 running in 5000 KB of RAM. SICStus 3 was version 3.11.0, GNU Prolog was version 1.2.16 and SWI Prolog was version 5.2.0.

Some tests could not be performed on some of the implementations: GNU Prolog does not support dynamic predicates, and Tricia and MacProlog proved incapable of consulting or compiling the *nrev(2)* benchmark.

The test machine was a 550MHz Macintosh PowerBook G4 running Mac OS X 10.3.1 in 512MB RAM, hosting Classic Mac OS 9.2.2.

The first two rows of the table show Open Prolog's performance in two situations. The first is where a garbage collection is performed in between consulting and executing the benchmark code, and the second row shows the result if a garbage collection is not performed. The difference between the two—about 7%—is due to an optimisation performed at the end of garbage collection. After garbage collection, the call action procedures for all procedures that do not still contain retracted clauses are reset to avoid performing a liveness test, since it is known at that time that all clauses are live.

The results show that Open Prolog offers high performance compared to other classic implementations. Open Prolog's single mode of operation compares very well with compiled Tricia, interpreted SICStus 2, compiled SICStus 2 for dynamic code and MacProlog32. In fact, Open Prolog is the fastest implementation that offers dynamic code facilities. It performs less

well against static compiled SICStus 2 code, being a little over half as fast, and is less than half as fast as static optimised MacProlog32 code.

As a 'classic' implementation Open Prolog's performance must suffer the overhead of emulation on the MC68LC020 by comparison with applications running directly on the host processor. Even when compared directly with Prolog implementations running directly on the host, however, Open Prolog continues to display a considerable speed advantage over interpreted or dynamic code. Against compiled code, it is a little slower than GNU Prolog and a little faster than SWI Prolog. Against SICStus 3 compiled static code, however, Open Prolog is much slower.

9.1.3 The Harness Benchmarks

This collection of benchmarks—so named because they are invoked by a 'harness' of test code in the file `harness.pl`—comprise eight separate benchmarks. In contrast to the *nrev* benchmark, no attempt is made to compensate for the overhead incurred in iterating the benchmark—the result is simply the elapsed time to perform the benchmark, overhead included.

The benchmark programs are *crypt* by Peter Van Roy, *deriv*, *qsort* and *query* by David H.D. Warren, *poly* by Ralph Haygood, *tak* by Evan Tick, with *primes* and *queens* of unknown authorship.[4]

9.2 Benchmark Results

Table 9.2 contains benchmark data for Open Prolog, Tricia, SICStus Prolog versions 2 and 3 both compiled and interpreted, SWI Prolog and GNU Prolog.

Results are similar to the results for the naive reverse benchmark: Open Prolog compares well with the other Prologs, apart from the compiled version

[4]According to the SICStus web site, these benchmarks were used in the Mercury project.

	Execution Time in Milliseconds							
Implementation	crypt	deriv	poly	primes	qsort	queens	query	tak
Open Prolog	27.3	0.12	65.2	2.8	1.01	665	5.5	339
Tricia (C)	100.6	0.695	320.6	10.7	4.39	2459	20.5	1192
MacProlog32 (Opt)	168.8	0.372	179.5	17.8	5.02	4136	27	1835
MacProlog32	202	0.785	292.3	23.4	7.67	5335	27.5	2495
SICStus 2 (C)	19.5	0.126	47.3	1.5	0.57	312	4.9	126
SICStus 2 (I)	120.4	0.756	304.4	14.3	5.37	2965	25.3	1837
GNU Prolog	18	0.123	51	1.8	0.8	432	3.6	199
SWI Prolog	23.9	0.126	64.1	2.1	0.83	556	4.6	234
SICStus 3 (C)	4.8	0.045	16.4	0.4	0.21	96	1.4	36
SICStus 3 (I)	42.7	0.245	108.2	4.7	1.61	1011	7.8	573

Table 9.2: Some benchmark performances for Prolog implementations running on a 550 MHz Macintosh PowerBook G4. The figures are milliseconds of elapsed time. No predicates are declared to be dynamic. Implementations above the double horizontal line are 'classic' implementations written in Motorola 68000 or 68020 code; these are executed by an MC68LC020 emulator built into the Mac OS. The other implementations run directly on the computer's PowerPC processor. Open Prolog is the fastest implementation that offers dynamic code facilities, but is slower than compiled static SISCtus code.[6]

of SICStus 3. Open Prolog is faster than any interpreter, while offering similar facilities. In addition, it is usually within a factor of two of SICStus Prolog 2 (Compiled), and is somewhat closer in speed to GNU Prolog and SWI Prolog; SICStus Prolog 3 (Compiled), however, is much faster, being between about three times (*deriv*) and about nine times (*tak*) faster.

9.3 Profiling

A special version of Open Prolog incorporates *profile vectors* of usage and 'hit' counters for interesting sections of code. A *usage counter* is incremented every time the section of code it is associated with is used. A *hit counter* work as follows: a special routine interrupts normal program execution at preset intervals, (perhaps every millisecond). The function of the interrupt is to increment the hit counter associated with the code section that was

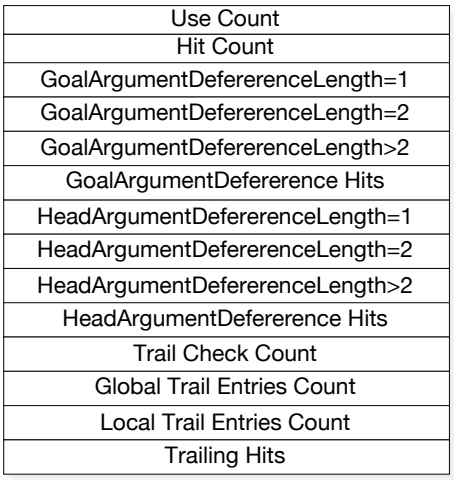

Figure 9.2: Layout of a Profile Vector. The profiling version of Open Prolog records information about sections of implementation code. Each section has an associated profile vector, the layout of which is depicted. The fourteen fields are used to record usage counts, hit counts, dereferencing behaviour and trailing behaviour. Sections of Open Prolog that are profiled include the code for every OPAM instruction and certain important subroutines.

interrupted, if any. Over a relatively long period, and assuming that the sampling process is independent of the operation of the program being profiled, the value in a hit counter reflects the amount of time the code section it corresponds to was active. The total execution time of a code section is its hit count multiplied by the sampling interval. Dividing the total execution time by the usage count gives the code section's average execution time. Dividing it by the total program execution time gives the proportion of total execution time the code section takes up, and thus its relative significance for the overall speed of the program.

The machine code for each individual OPAM instruction and some important subroutines are profiled in Open Prolog. A full *profile vector* of data is gathered for each code section. A profile vector includes information about head- and goal-argument dereferencing and trailing behaviour, as shown in

Figure 9.2.

Operation

While the profiling version of Open Prolog is running, usage and hit counters are updated all the time—the profiler is 'free-running'. To gather profile data for, say, the execution of a sample program, the *change* due to execution of the sample program is recorded by bracketing the call to it with two profiler control calls. The first call, to `'start$profile'/0`, copies the contents of each entry of every profile vector into a results buffer. After the program has been executed, a call to `'stop$profile'/0` subtracts the current value of each entry of every profile vector from its previously-captured counterpart in the results buffer. This yields the negative of the change in each count due to the execution of the sample program.

Another built-in profiling predicate, `'get$profile$data/2'`, gives access to the profile information gathered in the results buffer. The first call to this predicate returns the name and profile vector of a section of code as its first and second arguments. Each retry returns the 'next' profile vector until all profile vectors have been returned, at which point the predicate fails.[7]

Two other profiling built-in predicates are used. The predicate `'profile$sample$interval'/2` unifies the first argument with the current sampling interval and sets the new sampling interval to the second argument. The interval is understood as milliseconds if the argument is positive and microseconds if the argument is negative. Finally, `'get$profile$tick$count'/3` unifies the first argument with the number of samples taken when a profiled code section was interrupted (i.e. a *hit*), when Open Prolog was not the current process according to the Mac OS (a *miss*), and thirdly when another

[7]The order in which the profile vectors is returned depends on the order in which the code sections was assembled.

part of Open Prolog was interrupted (i.e. a *nil*).[8]

Summarising, the profiling predicates are:

- `'start$profile'/0`
- `'stop$profile'/0`
- `'get$profile$data/2'`
- `'profile$sample$interval'/2`
- `'get$profile$tick$count'/3`

Setting up a Profile Run

Given a minimum realistic sampling interval of the order of hundreds of microseconds and likely code section execution times of the order of microseconds, programs typically need to be executed many times to accumulate enough 'hits'. To prepare for a profile run, therefore, a clause is constructed with a body containing a sufficient number of calls to the code to be profiled, preceded by `'start$profile'/0` and followed by `'stop$profile'`. For example, to profile five runs of the naive reverse benchmark `nreverse`, the clause body to be generated would be:

```
...
'start$profile',nreverse,nreverse,
nreverse,nreverse,nreverse,'stop$profile',
...
```

One problem with this scheme is that the garbage produced during each run accumulates and may trigger a garbage collection during one of the runs.

[8]The idea was that the ratio of the sum of *hits* and *nils* to *misses* might give a rough idea of the proportion of processor time allocated to Open Prolog. In practice, the results didn't seem to make much sense, possibly because the Classic Mac OS was itself running in effectively a virtual machine and could be swapped out without being aware of it, though this is just speculation.

The chance of this happening can be minimised by allocating more memory. In any case, it does not affect the recorded usage and hit counts of those code sections that are not involved in memory management.

Recording Profile Data

The emulation code for every OPAM instruction and a number of important code sections were profiled in Open Prolog, amounting to 112 code sections in all. A profile vector of fourteen items was gathered for each section, yielding a matrix comprising 112 rows each of 14 data items generated by each profile run. Fortunately, in many cases the matrix is very sparse. Code sections with zero usage counts can be omitted, and trailing activity and head- and goal-dereferencing behaviour data can be omitted if there is zero usage.

9.3.1 Profile Interpretation—An Example

As an example, we take the extended code example from section 4.9 on page 111. This is identical to the naive reverse benchmark, except that the list contains just two elements to reduce the amount of code to be considered. Table 9.3 shows the basic information recorded during the profile run, where hits were recorded at 200 microsecond intervals.

Each row contains profile vector data for the code section named in the first column, (which must have been used at least once to be listed). The second and third columns contain the usage and hit counts for each section. The remaining columns contain goal- and head-variable dereferencing information and trailing activity. Head dereferencing information can include the number of times a dereference occurs of length 1 ('HD1'), length 2 ('HD2') or more than 2 ('HD>2') dereferences, and a hit count ('HH'). The same information can be recorded for goal dereferencing, in columns labelled 'GD1', 'GD2', 'GD>2' and 'GH' respectively. Trailing activity information can include the

Name	Uses	Hits	GD1	GH	HD1	HH	TA	TH
expand%	1	0	0	0	0	0	0	0
juggle%	1	28	0	0	0	0	0	0
call%	7501	44	5000	0	0	0	0	0
lastCall%	10000	72	7500	0	0	0	0	0
proc%	1	2	0	0	0	0	0	0
neck%	10000	28	0	0	0	0	0	0
procExit%	1	0	0	0	0	0	0	0
neckfoot%	7500	13	0	0	0	0	0	0
globalGlobal	2500	9	0	0	0	0	0	0
globalAtom	2500	8	2500	0	0	0	2500	0
varLocal	2500	5	0	0	0	0	0	0
varAtom	7500	15	7500	0	0	0	0	0
varStructure	7500	44	7500	0	0	0	2500	0
varVar	2500	7	0	0	2500	0	0	0
varRefL	5000	11	5000	0	5000	0	5000	0
voidGlobal	2500	6	0	0	0	0	0	0
atomLandVarLand	5000	11	0	0	5000	0	5000	0
integerVar	5000	9	0	0	5000	0	5000	0
structureGlobal	2500	7	0	0	0	0	0	0
structureLocal	2500	6	0	0	0	0	0	0
structureStructure	2500	12	0	0	0	0	0	0
structureLandVarLand	2500	9	0	0	2500	0	2500	0

Table 9.3: Profile results for 2,500 iterations of the extended code example from section 4.9, which is identical to the naive reverse benchmark, except that the list contains just two elements. Column 'GD1' is the number of goal variables dereferenced with a reference chain of length 1. 'GH' is the number of hits associated with goal dereferencing. Similarly 'HD1' is the number of head variables dereferenced with a reference chain of length 1 and 'HH' is the number of hits associated with head dereferencing. 'TA' is the number of times a check was made to determine if a unification needed to be trailed, and 'TH' is the number of hits associated with trailing activity. The size of the table is reduced by omitting profile data for unused code sections. Also omitted are columns for goal- and head-variable dereferencing activity where the dereference chain was greater than one, and the columns of numbers of global and local variables trailed, since no such activity was recorded. Hits were counted at 200 microsecond intervals.

number of times a trail was attempted ('TA'), the number of global and local variables actually trailed in the Trail Stack ('TG' and 'TL') and the number of hits ('TH') recorded while trail activity was in progress. Dereferencing and trailing activity columns may be omitted if they only contain zero values.

Each code section, except for `juggle%`, is the emulation code for one OPAM instruction, and is named after it. The `juggle%` code section contains the juggle code explained in Section 4.10 that implements the user interface while the OPAM is running.

It will be recalled that the code to be profiled is preceded by a call to `'start$profile'/0` and followed by a call to `'stop$profile'/0`.

The `'start$profile'/0` predicate preceding the calls to `nrev` effectively begins recording the profile data by copying the usage and hit counts. The predicate is entered via a `proc` instruction, but this instruction's use is recorded before the usage and hit counts are copied, so it is not counted in the difference between the before-and-after counts calculated by `'stop$profile'/0`. By contrast, a hit recorded during `'start$profile'/0` itself subsequent to the copying of the relevant hit counters *is* included in the profile. The predicate exits by executing a `procExit%` instruction—the only such instruction in the table.

Similarly, the start of the `'stop$profile'/0` call is profiled and included in the profile because profiling effectively continues until the differences between the counter values and the corresponding values stored in the buffer are actually calculated; this happens after execution has begun, i.e. after the predicate has executed the OPAM instructions `call%`, `expand%`, `proc%`.

The profile data can readily be processed to obtain the average execution time per code section and the proportion of execution time spent on any code section. Table 9.4 shows this information derived from the usage and hit counts in Table 9.3. The fourth column is an estimate of the average execution time, in nanoseconds, of each code section, and is calculated as

the number of hits multiplied by the interval between them (200 μseconds) and divided by the usage count. The final column is the percentage of total execution time taken by each code section; this is the ratio of the number of hits over the total number of hits.

It appears from Table 9.4 that the Mac OS and user interface code amounts to about 8% of overall execution time. Almost half (42%) of the execution time is taken up by three instructions: `call%`, `lastCall%` and `neck%`, and it would seem that the performance of these instructions is key to overall performance.

9.3.2 Profile Results

Profiles were generated of benchmark programs using the profiling code reproduced in Appendix B.4 on page 244. This code is based on the principles described in the previous section, and modelled on the code used in the *nrev2* benchmark in section 9.1.1.

The programs profiled were *nrev2*, (benchmarked on page 192), *tak*, (benchmarked on page 197), and *boyer*, a benchmark program due to Evan Tick from Lisp code by R. P. Gabriel (listed on page 249). Profiles were carried out under the same conditions as the benchmarks reported in section 9.1, i.e. a 550 MHz Macintosh PowerBook G4 running Mac OS X 10.3.1 in 512MB RAM, with Classic Mac OS 9.2.2.

The *nrev2* Profile

Beginning with the naive reverse benchmark, Table 9.5 presents usage and hit count data for 1000 runs of the naive reverse program. Average execution time and the proportion of overall execution time figures are derived from usage and hit count.

Name	Uses	Hits	nS	%
expand%	1	0	0	0
juggle%	1	28	5,600,000	8
call%	7501	44	1,173	13
lastCall%	10000	72	1,440	21
proc%	1	2	400,000	1
neck%	10000	28	560	8
procExit%	1	0	0	0
neckfoot%	7500	13	347	4
globalGlobal	2500	9	720	3
globalAtom	2500	8	640	2
varLocal	2500	5	400	1
varAtom	7500	15	400	4
varStructure	7500	44	1,173	13
varVar	2500	7	560	2
varRefL	5000	11	440	3
voidGlobal	2500	6	480	2
atomLandVarLand	5000	11	440	3
integerVar	5000	9	360	3
structureGlobal	2500	7	560	2
structureLocal	2500	6	480	2
structureStructure	2500	12	960	3
structureLandVarLand	2500	9	720	3

Table 9.4: Profile of 2,500 runs of the Extended Example program on page 112. The first column gives the name of each code section profiled; except for `juggle%`, code sections are named after the OPAM instructions they implement. (The juggle code time is effectively the time taken by the Mac OS and the user interface while the OPAM is running.) The second column is the usage count and the third column is the hit count. The fourth and fifth columns are calculations based on the usage and hit counts: the fourth column is an estimate of the execution time of each code section in nanoseconds, and the last column is an estimate of the percentage of total execution time taken up by each code section.
One `procExit%`, an `expand%`, a `proc%` and a `call%` instruction are artefacts and should be ignored.

Name	Uses	Hits	nS	%
expand%	1	0	0	0
juggle%	15	528	7,040,000	5
call%	31,001	194	1,252	2
lastCall%	466,000	2,787	1,196	25
proc%	1	2	400,000	0
neck%	466,000	1,083	465	10
procExit%	1	0	0	0
neckfoot%	31,000	61	394	1
globalGlobal	29,000	39	269	0
globalAtom	1,000	11	2,200	0
varGlobal	406,000	716	353	6
varLocal	29,000	43	297	0
varAtom	31,000	54	348	0
varStructure	899,000	2,582	574	23
varVar	435,000	1,234	567	11
varRefL	30,000	106	707	1
varLandVarLand	406,000	1,538	758	14
voidGlobal	1,000	7	1,400	0
atomLandVarLand	30,000	90	600	1
integerVar	30,000	123	820	1
structureGlobal	29,000	51	352	0
structureLocal	1,000	8	1,600	0
structureStructure	1,000	13	2,600	0
structureLandVarLand	29,000	82	566	1
Totals	3,381,019	11,352		100

Table 9.5: Profile of 1000 runs of the naive reverse benchmark program. The first column gives the names of each code section profiled; except for `juggle%`, code sections are named after the OPAM instructions they implement. The second column gives the number of times each code section was used. The third column records this number of hits (recorded at 200 μsecond intervals) for each code section. The fourth column is an estimate of the execution time of each code section in nanoseconds. The fifth column is an estimate of the percentage of total execution time taken up by each code section.
The OPAM instruction `procExit%` is executed at the end of the `'start$profile'/0` call and the instructions `expand%` and `proc%`, along with one `call%`, are executed at the start of the `'stop$profile'/0` call; they should be ignored. The time associated with the `juggle%` code section is used by the Mac OS User Interface.

Interpretation The profile of the naive reverse program shows that the overhead for user interface activity was about 5% of elapsed time. More importantly, it shows that approximately 40% out of the remaining 95%—almost half the time—is spent by the `call%`, `lastCall%`, and `neck%` instructions, which are concerned with locating the correct clause and constructing a local frame for it. In any search for improved performance, these are the areas that should pay dividends.

The *tak* Profile

The `tak` benchmark is a small benchmark generating many calls to arithmetic built-in predicates. Table 9.6 lists profile results for ten runs of the *tak* benchmark.

Interpretation The `tak` profile shows that over half the execution time was taken by code associated with built-in predicates, all of which are implemented as 'fast built-in predicates'. The asterisked entries show the number of such predicate calls (`'**fastBips'`) and the number of hits incurred. The `**evaluator` code is executed when an expression structure needs to be evaluated. The `**getNextArg` and `**getNextArgInteger` entries relate to subroutines used to dereference arguments to built-in predicates. Thus, the four bottom rows, accounting for over 50% of execution time, are associated with the execution of the build-in predicates `=</2`, `=/2`, `>/2` and `is/2`. It is clear that performance on the `tak` benchmark could be improved by further speeding up built-in predicates like these.

The *boyer* Profile

The `boyer` benchmark, shown in Table 9.7, is interesting because it is a larger program. It uses a number of built-in predicates heavily, viz. `arg/3` and `functor/3`.

Name	Uses	Hits	nS	%
expand%	1	0	0	0
juggle%	48	1,111	4,629,167	3
call%	1,749,241	4,393	502	12
lastCall%	636,100	1,622	510	4
cut%	477,070	2,290	960	6
efailEntry	159,020	374	470	1
proc%	1	0	0	0
neck%	795,120	1,487	374	4
procExit%	1	0	0	0
globalGlobal	611,100	1,044	342	3
varGlobal	1,163,060	2,050	353	6
varLocal	795,100	1,169	294	3
refLGlobal	134,040	263	392	1
refLLocal	477,060	366	153	1
voidGlobal	20	0	0	0
integerGlobal	40	0	0	0
integerLocal	20	0	0	0
**fastBip	1,749,240	11,419	1,306	32
**evaluator	477,060	4,607	1,931	13
**getNextArgInteger	477,060	286	120	1
**getNextArg	3,021,420	3,602	238	10

Table 9.6: Profile of 10 runs of the tak benchmark program. The layout and significance of the columns is the same as for the naive reverse profile, Table 9.5. The four entries on the bottom row are for sections of code associated with the built-in predicates `=</2`, `=/2`, `>/2` and `is/2`.

Name	Uses	Hits	nS	%
expand%	1,537,631	2,524	328	1
orCall%	387,050	3,990	2,062	1
juggle%	537	15,354	5,718,436	4
call%	6,635,651	44,496	1,341	11
lastCall%	1,092,850	9,773	1,789	2
cut%	951,270	10,547	2,217	3
fail%	507,950	4,176	1,644	1
proc%	1,537,631	35,817	4,659	9
neck%	2,427,600	10,253	845	3
foot%	951,270	2,030	427	1
neckCutFoot%	384,530	2,129	1,107	1
localLocal	1,150,440	2,774	482	1
varGlobal	1,844,600	5,660	614	1
refLGlobal	1,152,360	5,577	968	1
refLLocal	2,867,550	5,385	376	1
refLRefL	950,150	3,753	790	1
**fastBip	4,146,890	64,345	3,103	16
**lookup name	766,960	126,941	33,102	32
**evaluator	940,560	19,592	4,166	5
**getNextArgInteger	2,821,680	4,722	335	1
**getNextArg	3,571,480	4,903	275	1

Table 9.7: Profile of a run of the `boyer` benchmark program. Only entries accounting for 1% or more of the execution time are listed; otherwise the layout and significance of the columns is the same as for the naive reverse profile, Table 9.5. The five entries on the bottom row are for sections of code associated with the built-in predicates, mainly `arg/3` and `functor/3`.

Interpretation Similarly to the *tak* profile, the *boyer* profile shows that a great deal of time—over 50%—is spent executing built-in predicates. In particular, the code section entitled `**lookup name` accounts for almost one third of overall execution time.[9] It turns out that this is due to the frequent use of the `functor/3` built-in predicate, which associates an atom and an arity with a term having a functor with the same name as the atom but with the arity given. In Open Prolog, to make the association between, say, the atom and the functor, it is necessary to retrieve the atom's spelling, combine it with the arity and use it to locate the functor's entry using the hash table. This is a very costly operation, but it could be avoided by modifying the Name Table to associate all atoms and functors that have the same spelling. This would significantly improve Open Prolog's performance in programs such as the *boyer* benchmark.

[9] A similar result was observed for the `boyer` benchmark with BIM Prolog [27].

Chapter 10

Conclusions & Further Work

10.1 General Conclusions

As stated in the introduction and again in Chapter 4, the aim of the work described here was to build an implementation of Prolog that would combine the advantages of compiler-based implementations—speed and memory efficiency—with those of interpreter-based implementations—ease of inspection of programs, ease of modification and ease of debugging—in one mode of operation.

Open Prolog shows that a conceptually simple structure-sharing implementation of Prolog can be built that is compact and reasonably fast.

Control constructs, catch-and-throw exception signalling, 'logical' database semantics, a variety of external predicate interfaces and a form of interrupt handling have all been incorporated without significant loss of efficiency. The 'dual-PC' approach to executing image machine instructions appears to offer a simple and orderly way to organise the emulation.

10.2 Speed

The comparisons between Open Prolog and existing implementations on the Macintosh show that, despite running on a MC68LC020 emulator, Open Prolog is fast, and is quite comparable to SWI Prolog and GNU Prolog running directly on the host processor. Compared to SICStus Prolog 2 (running on the MC68LC020 emulator), Open Prolog is about half as fast, and SICStus Prolog 3, running on the host processor directly, is between three and nine times faster. Of all implementations tested offering dynamic code facilities, Open Prolog is fastest.

The profiling analysis shows that a high proportion of the execution time is spent in the `call`, `lastCall` and `neck` instructions, all instructions that are connected with selecting a clause to execute and then constructing an instance of it. In particular, `call` and `lastCall` are long and frequently used instructions. In many cases of interest, the WAM avoids these instructions (or their equivalents) completely. For example, in the `append/3` code, the WAM uses indexing to avoid a large overhead in selecting a clause, and avoids having to construct an environment for the last call; thus, the main business of the `call` and `neck` instructions in the OPAM is sidestepped in the WAM. Looking rather crudely at the profile data in Table 9.5 on page 207, if the `call`, `lastCall` and `neck` instructions—those instructions concerned with searching for clauses and constructing environments—were removed, execution time would be reduced by more than one third. See section 10.4.3 on page 218 for a preliminary discussion as to how this might be approached.

The profiling results also show considerable room for improvement in the speed of built-in predicates, which show up in the `tak` and `boyer` benchmarks. Another problem specific to the OPAM is related directly to the dual-PC mode of execution. In the absence of branch instructions, it would be possible to replace a fixed sequence of instructions, separately compiled, with an

equivalent sequence of instructions to do the same thing more quickly. Now, this approach, commonly taken in compilers, is unsuitable for Open Prolog, since the separate identity of the individual instructions, and hence the correspondence between the source code and the machine code, is normally lost. However, if it was suitable, say for private code, it still isn't possible in the OPAM because all that is available is a sequence of instruction *fragments*. Since not all the instructions are available, the scope for optimisation seems limited.

10.3 Contributions

The principal contribution is a design for a Prolog implementation based on relatively simple principles that combines high speed compilation and reasonable runtime performance with the ability to modify, inspect and debug Prolog code.

Influenced by principles of Direct Correspondence Architectures, the design is based on an abstract machine with an instruction set such that every aspect of the source code is represented in the machine code. This facilitates the the reconstruction of the source code, for inspection and debugging purposes. It also simplifies compilation.

Given the special requirement that the conformation of the original clauses was to be preserved, for listing, debugging and assertion/retraction, the following innovations are employed:

- Dual PCs are used to allow the OPAM code to be simplified. The use of dual PCs facilitates a more straightforward representation of the code for each clause, and the core of the OPAM interpreter is simplified by this design feature. It is necessary to make the interpreter switch between dual- and single-PC modes; this is accomplished as a side-effect to the appropriate instructions.

- Control constructs such as disjunction and if-then are represented specially so that their constituent goals are compiled at the same level as regular goals in a clause body. Combined with Non-Removable (NR) frames and Catch Chains, this facilitates fast implementation of control constructs while ensuring that the interplay between control constructs and the cut is correctly and flexibly handled.

- The number of tags used to represent Prolog terms is almost doubled by adding 'landed' variants. However, incorporation of landed tokens at the ends of structures reduces the amount of stack space required and speeds access to right-tail-nested structures.

- The application of Lindholm & O'Keefe's 'logical semantics' is limited to clauses that have been modified since the last garbage collection. This is made possible by using specialised call action procedures for modified and unmodified procedures. This appears to improved performance by about 7%.

- Garbage collection of retracted clauses is facilitated by the addition of non-executable clause-crawling data to the OPAM code of each clause. This data allows the garbage collector to locate the start of the clause while scanning forward from an embedded skeleton.

- A variant of reference chain cleanup or *shunting* is used to avoid retaining intermediate references that are otherwise unused.

Finally, the actual implementation was robust and complete enough to be widely used, it has offered a tested that demonstrates the traditional virtues of interpretation—the ability to modify Prolog programs by asserting and retracting clauses, and the ability to inspect, list and debug clauses—can be had while still having a reasonably high performance. Results from profiling

reveal bottlenecks that could be relieved with minimal changes to Open Prolog's design and that would considerably improve its performance in certain benchmarks.

10.4 Future Work

10.4.1 Reimplementation & Porting

The most immediate requirement is for Open Prolog to be ported to Mac OS X. As previously mentioned, much of the present implementation is written in 68000 Assembly Language with some written in Pascal. When running on a Mac OS X based machine, it runs on a Motorola MC68LC020 emulator in the 'Classic' Mac OS environment. To port Open Prolog so that it runs in the Mac OS X environment, directly on the host processor so that it does not need an MC68000 emulator, it will essentially have to be rewritten. It is not clear how much Open Prolog suffers from running on the MC68LC020 emulator as opposed to running natively. A rather crude attempt to quantify this, by writing very simple dispatch code in C++ and compiling it into 68000, 68020 and generic PowerPC code shows a ratio of approximately 2.5 and 2 to 1 in execution time. However, as the execution code of Open Prolog is hand-written in 68000 assembly language, it is not clear that that factor of improvement is even attainable, since hand-written PowerPC code is difficult to write well. Nevertheless, there appears to be some scope for improvement. Since Apple has very recently announced a transition to Intel processors, the target of the re-implementation will likely be the Intel X86 architecture.

10.4.2 Implementation Improvements

Clearly, the profile results point towards areas in which improvements could be made: built-in arithmetic and name-handling predicates in particular.

The point is made in [43, p9] that high speed built-in predicates are important to the real-world performance of a Prolog implementation. One obvious way to improve performance in built-in predicates is to enrich the instruction set with, for example, a selection of heavily used special-case built-ins. Along the same lines, a more direct dispatch mechanism for built-in predicates, bypassing the `call...` instructions completely would undoubtedly improve performance. In [43], consideration is given to reducing the amount of operand type checking necessary. A simple analysis of a sequence of goals might reveal where special variants of built-ins with reduced or absent operand type checking could be used safely. For instance, a predicate using an argument assigned a value by the `is/2` predicate can be guaranteed that the argument is an integer.

Following on from that idea, recall that structures are are always constructed as molecules, and variables within structures are always global. However, there may be circumstances where it can be ascertained that the structure will never be accessed from outside the scope of the clause instance in which it is generated. In that case, the structure variables can be local. Moreover, if it can only be accessed from its parent clause instance, it does not need to be represented as a molecule; a pointer to its skeleton would be sufficient. This idea of a *local structure* would likely save memory and improve execution speed.

There are some obvious improvements possible from removing branch instructions from call instructions. The original intention here was to make call instructions as similar as possible to structure literals, since they both represent terms. A similar 'linearising' approach to structure literals, and possibly the addition of a special 'list' literal might also pay speed dividends. This would be similar in spirit to Cdr Coding, a technique used in Lisp implementations.

A much more radical experiment would be to redesign Open Prolog as a

structure copying implementation, but along the same principles otherwise. It would be most interesting to see how well the two implementations would compare.

10.4.3 Architectural Improvements—Local Frame Reuse

In the following paragraphs will be outlined a suggestion for improving the speed and memory management of Open Prolog which would be a very interesting project. In a nutshell, the technique is of local stack frame *reuse*, as distinct from *reallocation*. The distinction is that in the case of reallocation, memory is recovered and then reallocated, but the contents of the memory cells in question is not used. By contrast, reuse also makes use of some or all of the contents of the memory cells, reducing the amount of initialisation required. In related work, Köves and Szeredi discuss the potential for identifying situations where stack frames can be reused for passing arguments [43, p4]. Debray discusses environment reuse in [26] and cites work by Meier [50]. Zhou explores reuse in the context of ATOAM, a development of the WAM, in [78].

Local Frame Reuse

If a determinate last call is about to be made, and if certain assumptions can be made about the disposition of local variables, then the caller's local stack frame can be progressively turned into the callee's frame, possibly reusing some of the information contained in the caller's frame.

Consider the unification of arguments in the last call depicted in Figure 10.1. The diagram shows the sequence of events that occurs when a three-argument last call is made. In each of the four steps in the sequence, the goal's arguments are on the left and the clause instance's head arguments are on the right. An argument in grey is part of the callee's environment.

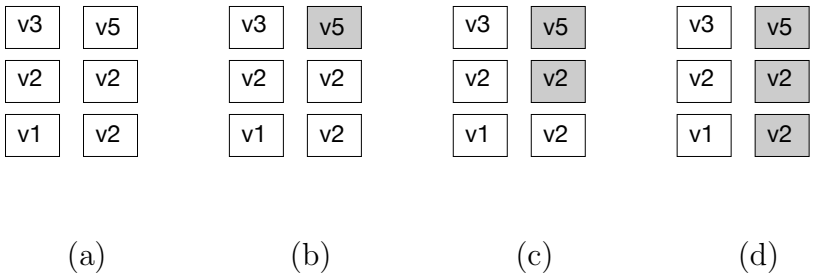

Figure 10.1: This sequence of four diagrams depicts the sequence of events when a three-argument last call is made. In each diagram, the goal's arguments are on the left and the clause instance's head arguments are on the right. An argument in grey is part of the callee's environment. Higher numbered variables are 'newer' than, and could therefore potentially contain references to, lower numbered variables.

Assume that all the arguments are local variables and that higher numbered variables are newer than lower numbered variables. In what follows, all references will be to local variables in the frame that is being reused. The term 'head variable' means a local variable as a head argument; likewise the term 'goal variable' refers to a local variable in a goal argument. Unification of the arguments begins at the top and moves downwards. Once the first head argument, `v5`, is unified with the first goal argument, `v3`, it becomes part of the callee's environment. When the next head argument, `v2`, is unified with its corresponding argument, `v2` (i.e. with itself—a null operation), it too becomes part of the new callee's environment. It must be noted that newer local variables such as `v3` might reference `v2`. Therefore, one rule of the transition process is that caller variables that are newer than the variables that have become part of the callee's environment are unsafe. The transition continues as the arguments are successively unified, until, at the end, the local stack frame belongs completely to the callee.

Conditions for reusing a Local Stack Frame

The conditions under which a progressive transition from caller frame to callee frame occur ensure that, when a head argument is being unified with a goal argument, variables referenced in the goal argument are guaranteed to be part of the caller's section of the frame—they must not be newer than the variable in the head argument and they must not reference older variables that are already part of the callee's environment. To ensure these conditions are met, the following must hold:

- A local variable appearing as a head argument must never be older than a local variable in its corresponding goal argument,
- If a local variable appears as a goal argument, it must do so before any older local variables appear as head arguments.

The compiler can attempt to arrange variable allocation to fulfil these conditions. If it is unable to make such an allocation, it can reallocate variables as global until the conditions are met for the remaining variables.

Unbound Local Variables

One problem not addressed so far is what happens if a head variable dereferences to an unbound goal variable. The head variable would be bound to a variable in a frame that was about to disappear. The solution is to reverse the direction of referencing in a case like this—that is, if a head variable dereferences an [older] goal variable, then the goal variable should be made point to the head variable, which should be unbound. Later on, when the caller's environment is phased out, the goal variable will be effectively deleted, leaving the head variable unbound, as required. This solution also works where two or more head variables are bound to the same unbound goal variable. The first binding will result in the goal variable pointing to

the first head variable; the second binding will dereference the goal variable to the first head variable and thus bind the two head variables together as required.

The biggest problem may be the limited applicability of this technique. In order to comply with the conditions for reusing a stack frame, the compiler has to have knowledge of both the caller and the callee. In the context of a clause-by-clause compiler, such as Open Prolog, this means that the caller clause and the callee clause must be the same—that is, the technique can readily be applied to determinate tail recursive calls . This may not be such a drawback as it seems, as many Prolog procedures are determinate and tail recursive.

The Benefits of Stack Frame Reuse

Firstly, the functions carried out by the `neck`, `call`, `lastCall` and `neck` instructions would be greatly simplified, leading to the elimination of some or all of them. It would be necessary to ensure the call was determinate and it would be necessary to update the global frame pointer, but no new environment would need to be generated.

Secondly, local variables could be used in last call arguments. The circumstances of their use might have to be restricted to allow the phased transition of the frame, but in practice a graceful fall-back is available since local variables can be reallocated as global if necessary.

Thirdly, scope exists for the 'nulling' of data movements. For example, if a local variable occupies the same argument number in the head as in the last call, then it doesn't have to be moved, and so the instructions for moving it can be omitted. This should result in a significant speedups in many cases of interest, just as it does in the WAM. It would be necessary to ensure that such variables were not unsafe, in the sense that they could not become bound to some other unbound variable in the frame. If this could be

done at compile time, then the full benefit of nulling would accrue.

Beyond the advantages alluded to, another possibility comes into view. First, if local variables in a fixed local frame are being re-used, they could easily be realised with processor registers. Second, if the same clause is being called by the same goal each time, the sequence of instructions executed is known, and might possibly be compiled as a sequence. This is similar to Krall and Neumerkel single-PC variant of the Vienna Abstract Machine (the VAM_{1P} [47]). Whereas the two-PC variant of the VAM (the VAM_{2P}) combines instructions at runtime, the VAM_{1P} arranges to combine them at compilation time.

Taken together, the possibility of register-based compilation of sequences of OPAM instructions arises. It would be very interesting indeed to explore and develop this possibility.

Bibliography

[1] Hassan Aït-Kaci. *Warren's Abstract Machine: A Tutorial Reconstruction.* MIT Press, 1991.

[2] J. Andersson, S. Andersson, K. Boortz, M. Carlsson, H. Nilsson, T. Sjöland, and J. Widén. SICStus Prolog User's Manual. Technical Report T83:01A, Swedish Institute of Computer Science, 1994.

[3] Jonas Barklund. A Garbage Collection Algorithm for Tricia. Technical Report 37B, Uppsala Programming Methodology and Artificial Intelligence Laboratory (UPMAIL), Uppsala University, 1987.

[4] Jonas Barklund and Håkan Millroth. Code Generation and Runtime System for Tricia. Technical Report 36, Uppsala Programming Methodology and Artificial Intelligence Laboratory (UPMAIL), Uppsala University, 1986.

[5] Gérard Battani, Henry Méloni, and René Bazzoli. Interpréteur du langage de programmation PROLOG. DEA report, Groupe Intelligence Artificielle, Faculté des Sciences de Luminy, Université Aix-Marseille II, Marseille, France, 1973.

[6] James R. Bell. Threaded code. *Communications of the ACM*, 16(6):370–372, 1973.

[7] David L. Bowen, Lawrence Byrd, Luís M. Pereira, Fernando C. N. Pereira, and David H. D. Warren. PROLOG on the DECSystem-10 User's Manual. Technical report, Department of Artificial Intelligence, University of Edinburgh, Scotland, 1981.

[8] Kenneth A. Bowen, Kevin A. Buettner, Ilyas Cicekli, and Andrew K. Turk. The Design and Implementation of a High-Speed Incremental Portable Prolog Compiler. In Ehud Shapiro, editor, *Third International Conference on Logic Programming*, volume 2, pages 375–388, London, July 1986.

[9] Robert S. Boyer and Jay S. Moore. The sharing of Structure in Theorem Proving Programs. In Bernard Meltzer and Donald Michie, editors, *Machine Intelligence*, volume 7, pages 101–116. Edinburgh University Press, Edinburgh, U.K., 1972.

[10] Mike Brady. 'Runsort'–An Adaptive Mergesort For Prolog. Technical Report TCD-CS-2005-34, Department of Computer Science, University of Dublin, Trinity College Dublin, Ireland, April 2005. `http://www.cs.tcd.ie/publications/tech-reports/reports.05/TCD-CS-2005-34.pdf`.

[11] Ivan Bratko. *Prolog Programming for Artificial Intelligence*. Addison Wesley, third edition, September 2000.

[12] Maurice Bruynooghe. An Interpreter for Predicate Logic Programs. Report CW 10, Department of Computer Science, Katholieke Universiteit Leuven, Belgium, October 1976.

[13] Maurice Bruynooghe. The Memory Management of PROLOG Implementations. In Sten-Åke Tärnlund and Keith L. Clark, editors, *Logic Programming*, pages 83–98. Academic Press, 1982. Originally in the Workshop on Logic Programming, Debrecen, Hungary, 1980.

[14] Maurice Bruynooghe. Garbage collection in Prolog interpreters. In Campbell [16], pages 259–267.

[15] Kevin A. Buettner. Fast Decompilation of Compiled Prolog Clauses. In Shapiro [60], pages 663–670.

[16] John A. Campbell, editor. *Implementations of PROLOG*. Ellis Horwood Series in Artificial Intelligence. Ellis Horwood, Chichester, England, 1984.

[17] Mats Carlsson. Compilation for Tricia and Its Abstract Machine. Technical Report 35, Uppsala Programming Methodology and Artificial Intelligence Laboratory (UPMAIL), Uppsala University, September 1986.

[18] Mats Carlsson. The SICStus Emulator. Technical Report T91:15, Swedish Institute of Computer Science, 1991.

[19] Mats Carlsson. Personal communication, 2003.

[20] Mats Carlsson. Personal communication, 2005.

[21] Keith Clark, Frank McCabe, N. Johns, and Clive Spenser. *LPA MacPROLOG Reference Manual*. Logic Programming Associates, London, England, 1988.

[22] William F. Clocksin and Christopher S. Mellish. *Programming in Prolog: Using the ISO Standard*. Springer, fifth edition, September 2003.

[23] Jacques Cohen. A View of the Origins and Development of Prolog. *Communications of the ACM*, 31(1):26–36, 1988.

[24] Alain Colmerauer. An introduction to Prolog III. *Communications of the ACM*, 33(7):69–90, 1990.

[25] Alain Colmerauer and Philippe Roussel. The birth of Prolog. In *The second ACM SIGPLAN conference on History of programming languages*, pages 37–52. ACM Press, 1993.

[26] Saumya K. Debray. A Simple Code Improvement Scheme for Prolog. *Journal of Logic Programming*, 13(1):57–88, 1992. Also available at `http://www.cs.arizona.edu/people/debray/papers/prolopt.ps`.

[27] Bart Demoen and André Marien. Can Prolog Execute as Fast as Aquarius. `http://www.cs.kuleuven.ac.be/~bmd/pubs/aquarius.ps`, 1992.

[28] Pierre Deransart, Laurent Cervoni, and AbdelAli Ed-Dbali. *Prolog: The Standard: Reference Manual*. Springer-Verlag, 1996.

[29] Robert B. K. Dewar. Indirect threaded code. *Communications of the ACM*, 18(6):330–331, June 1975.

[30] Daniel Diaz and Philippe Codognet. The GNU Prolog system and its implementation. In *Proceedings of the 2000 ACM symposium on Applied computing*, pages 728–732. ACM Press, 2000.

[31] T. P. Dobry. *A High Performance Architecture for Prolog*. Kluwer Academic Press, 1990.

[32] Vladimir Estivill-Castro and Derick Wood. A Survey of Adaptive Sorting Algorithms. *ACM Computing Surveys*, 24(4):441–476, 1992.

[33] Zsuzsa Farkas, Péter Köves, and Péter Szeredi. MProlog: an implementation overview. In Evan Tick and Giancarlo Succi, editors, *Implementations of Logic Programming Systems*, pages 103–117. Kluwer Academic Publishers, 1994.

[34] David Hillel Gelernter. *Machine Beauty: Elegance and the Heart of Technology*. Basic Books, New York, NY, USA, 1998.

[35] Steve Gregory. *Parallel Logic Programming in PARLOG: The Language and Its Implementation*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1987.

[36] Patricia M. Hill and John W. Lloyd. *The Gödel Programming Language*. MIT Press, 1994.

[37] James Hoch and Carla Bauer. The DELtran Project. In Veljko M. Milutinovic, editor, *High Level Language Computer Architecture*, pages 332–355. Computer Science Press, 1989.

[38] Joshua. S. Hodas. Compiling Prolog: From the PLM to the WAM and Beyond. Ph.D. Qualifying Paper, Department of Computer and Information Science, University of Pennsylvania, Fall 1990.

[39] International Standards Organisation. *ISO/IEC 132111:1995 Information technology -- Programming languages -- Prolog --Part 1: General core*. International Standards Organisation, Geneva, Switzerland, `http://www.iso.ch/`, 1995.

[40] Joxan Jaffar, Spiro Michaylov, Peter J. Stuckey, and Roland H. C. Yap. The CLP(R) Language and System. *ACM Transactions on Programming Languages and Systems*, 14(3):339–395, 1992.

[41] Richard Jones and Rafael Lins. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. John Wiley & Sons, Inc., New York, NY, USA, 1996.

[42] Donald E. Knuth. *Sorting and Searching*, volume 3 of *The Art of Computer Programming*. Addison-Wesley, Reading, MA, USA, second edition, 1998.

[43] Péter Köves and Péter Szeredi. Getting the Most Out of Structure Sharing. In *Collection of Papers on Logic Programming*, pages 69–84. SzKI, Budapest, 1988.

[44] Robert A. Kowalski. Algorithm = Logic + Control. *Communications of the ACM*, 22(7):424–436, 1979.

[45] Robert A. Kowalski. The early years of logic programming. *Communications of the ACM*, 31(1):38–43, 1988.

[46] Robert A. Kowalski and Donald Kuehner. Linear Resolution with Selection Function. *Artificial Intelligence*, 2:227–260, 1971.

[47] Andreas Krall and Ulrich Neumerkel. The Vienna Abstract Machine. In P. Deransart and J. Małuszyński, editors, *Proceedings of Programming Language Implementation and Logic Programming Second International Workshop PLILP'90*, volume 456 of *Lecture Notes in Computer Science*, pages 121–135. Springer, 1990.

[48] Xining Li. A New Term Representation Method for Prolog. *Journal of Logic Programming*, 34(1):43–57, 1998. Also available at `http://draco.cis.uoguelph.ca/article/jp02.ps`.

[49] Timothy G. Lindholm and Richard A. O'Keefe. Efficient Implementation of a Defensible Semantics for Dynamic Prolog Code. In J. L. Lassez, editor, *Proceedings of Fourth International Conference on Logic Programming*, Melbourne, 1987.

[50] Micha Meier. Recursion vs. Iteration in Prolog. In *Proceedings of the 1990 North American Conference on Logic Programming*, pages 63–79. MIT Press, 1990.

[51] Christopher S. Mellish. An Alternative to Structure-Sharing in the Implementation of A Prolog Interpreter. In Sten-Åke Tärnlund and Keith L. Clark, editors, *Logic Programming*, pages 99–106. Academic Press, 1982.

[52] Christopher D. S. Moss. Cut and Paste—defining the impure Primitives of Prolog. In Shapiro [60], pages 686–694.

[53] Richard A. O'Keefe. Samsort (Smooth Applicative Mergesort). `http://www.j-paine.org/prolog/tools/files/samsort.pl`, 1984.

[54] Richard A. O'Keefe. *The Craft of Prolog*. MIT Press, 1990.

[55] Fernando C. N. Pereira. C-Prolog User's Manual. Technical report, Edinburgh Computer Aided Architectural Design, University of Edinburgh, Scotland, August 1983. Also available at: `http://www.gtoal.com/athome/edinburgh/cprolog/cprolog.html`.

[56] Grant Maxwell Roberts. An implementation of PROLOG. Master's thesis, University of Waterloo, Department of Computer Science, Waterloo, Ontario, Canada, 1977.

[57] John A. Robinson. A Machine-Oriented Logic Based on the Resolution Principle. *Journal of the ACM*, 12(1):23–41, 1965.

[58] Philippe Roussel. *Definition et traitment de l'egalité formelle en démonstration automatique*. Thèse de troisième cycle, Faculté des Sciences, Université Aix-Marseille II, Luminy, France, 1972.

[59] Dan Sahlin and Mats Carlsson. Variable shunting for the WAM. Research report SICS/R91–07, Swedish Institute of Computer Science, 1991.

[60] Ehud Shapiro, editor. *Proceedings of Third International Conference on Logic Programming*, volume 225 of *Lecture Notes in Computer Science*. Springer-Verlag, 1986.

[61] Ehud Shapiro. A Subset of Concurrent Prolog and its Interpreter. In Ehud Shapiro, editor, *Concurrent Prolog: Collected Papers (Volume I)*, pages 27–83. MIT Press, Cambridge, MA, USA, 1987.

[62] Zoltan Somogyi, Fergus Henderson, and Thomas Conway. Mercury: an efficient purely declarative logic programming language. In *Proceedings of the Australian Computer Science Conference*, pages 499–512, February 1995.

[63] J. Michael Spivey. *An Introduction to Logic Programming through Prolog*. Prentice Hall Europe, Hertfordshire, England, 1996.

[64] Brian D. Steel. Personal communication, 2003.

[65] Leon Sterling and Ehud Shapiro. *The Art of Prolog: Advanced Programming Techniques*. MIT Press, second edition, March 1994.

[66] Péter Szeredi. The Early Days of Prolog in Hungary—a personal account. In *Association of Logic Programming Newsletter*. November 2004. `http://www.cs.kuleuven.ac.be/~dtai/projects/ALP/newsletter/nov04`.

[67] Paul Tarau. BinProlog: a Continuation Passing Style Prolog Engine. In Maurice Bruynooghe and Martin Wirsing, editors, *Proceedings of the Programming Language Implementation and Logic Programming, 4th International Symposium, PLILP'92*, volume 631 of *Lecture Notes in Computer Science*, pages 479–480. Springer, 1992.

[68] Andrew Taylor. Parma—Bridging the Performance GAP Between Imperative and Logic Programming. *Journal of Logic Programming*, 29(1-3):5–16, 1996.

[69] Evan Tick. *Memory Performance of Prolog Architectures*. Kluwer Academic Publishers, 1988.

[70] Peter Van Roy. *Can Logic Programming Execute as Fast as Imperative Programming?* PhD dissertation, University of California at Berkeley, Department of Computer Science, 1990. Revised version published as *Fast Logic Program Execution* by Intellect Books.

[71] Peter Van Roy. 1983–1993: The Wonder Years of Sequential Prolog Implementation. *Journal of Logic Programming*, 1994.

[72] David H. D. Warren. Implementing Prolog—compiling predicate logic programs. Research Report 39, Department of Artificial Intelligence, University of Edinburgh, Scotland, May 1977.

[73] David H. D. Warren. Implementing Prolog—compiling predicate logic programs. Research Report 40, Department of Artificial Intelligence, University of Edinburgh, Scotland, May 1977.

[74] David H. D. Warren. An Improved Prolog Implementation Which Optimises Tail Recursion. Research Paper 156, Department of Artificial Intelligence, University of Edinburgh, Scotland, 1980.

[75] David H. D. Warren. An abstract Prolog instruction set. Technical Report 309, Artificial Intelligence Center, SRI International, October 1983.

[76] David H. D. Warren. Optimizing Tail Recursion in Prolog. In Michel van Caneghem and David H. D. Warren, editors, *Logic Programming*

and its Applications, Ablex Series in Artificial Intelligence, pages 77–90. Ablex Publishing Corporation, Norwood, New Jersey, 1986.

[77] Robert G. Wedig. Direct Correspondence Architectures: Principles, Architecture, and Design. In Veljko M. Milutinovic, editor, *High Level Language Computer Architecture*, pages 305–331. Computer Science Press, 1989.

[78] Neng-Fa Zhou. Parameter passing and control stack management in Prolog implementation revisited. *ACM Transactions on Programming Languages and Systems*, 18(6):752–779, 1996.

[79] Aquarius Prolog Benchmark Suite. `ftp://ftp.info.ucl.ac.be/pub/aquarius/Benchmarks.tar.Z`, 1993.

[80] Free Online Dictionary of Computing. `http://www.nightflight.com/foldoc-bin/foldoc.cgi?Prolog`, 1993.

[81] LPA Mac Prolog. `http://www.lpa.co.uk/mac.htm`.

[82] Macintosh Programmer's Workshop (Program Development Software). `http://developer.apple.com/tools/mpw-tools/`, 1984.

[83] Naive Reverse Benchmark. `http://www.sics.se/isl/sicstus/bench.pl`, 1984.

[84] ResEdit. `http://developer.apple.com/documentation/mac/resedit/resedit-2.html`.

[85] SICStus Performance Benchmarks Page. `http://www.sics.se/isl/sicstuswww/site/performance.html`, 2004.

[86] SWI Prolog Home Page. `http://www.swi-prolog.org/`, 1987.

[87] Tricia FTP Site. `ftp://ftp.csd.uu.se/pub/Tricia/Mac/`, 1987.

Appendix A

PLM Instruction Set Summary

Clause Selection Instructions	
`enter`	Initialises the control information in a new environment—the contents of `VV`, `X`, `A`, `V1` and `TR` are copied into the local frame. Then `VV` and `VV1` are set to the values of `V` and `V1`, making the partly constructed new environment the current choice point.
`try(L)`	Updates the current instance's choice point data by storing the address of the next `try` or `trylast` instruction in the `FL` register. Execution is transferred to the clause code at label `L`.
`trylast(L)`	Ends use of the current instance's environment as a choice point by setting `VV` and `VV1` to the values they had when the current goal was called. Execution is transferred to the clause code at label `L`.

Table A.1: Clause Selection Instructions

Unification Instructions	
`uvar(N,F,I)`	Unify argument `N` with the first occurrence of variable `I` of type `F` (`local`/`global`).
`uref(N,F,I)`	Unify argument `N` with a second or later occurrence of variable `I` of type `F` (`local`/`global`).
`uatom(N,A)`	Unify argument `N` with atom `A`
`uint(N,I)`	Unify argument `N` with integer `I`
`uskel(N,S)`	Unify argument `N` with structure `S`. (The structure is encoded as a literal after the `foot` instruction.)

Table A.2: PLM Unification Instructions

Other Head & Neck Instructions	
`neck(I,J)`	Marks completion of unification of the current goal with the new clause instance. Turns the environment of the new instance into the environment for the new current goal by setting `X` and `X1` from registers `V` and `V1`. Registers `V` and `V1` are updated to point to the top of the stacks, beyond the new frames.
`init(I,J)`	Global variables `I` to `J` are initialised to `undef`.
`localinit(I,J)`	Local variables `I` to `J` are initialised to `undef`.
`ifdone(L)`	Used before level 1 argument instructions. If the structure unified with a reference, no unification of level 1 arguments is needed, so the level 1 instructions are skipped by transferring to label `L`.

Table A.3: Neck and Head Instructions

Body Instructions	
`call(L)`	Calls the procedure whose `enter` instruction is at label `L`. Sets register `A` to point to the goal argument literals and to the continuation, effectively the return address.
`cut(I)`	Discards all local frames newer than the current one, tidies up the Trail to remove redundant trail entries. Resets `VV` and `VV1` from the relevant fields of this clause's parent.

Table A.4: Body Instructions

Foot Instructions	
`foot(N)`	Returns to the caller. If the goal has executed determinately, the local frame is deallocated. Registers `A`, `X` and `X1` are reset from the local frame; thus control is transferred to the parent.
`neckfoot(J,N)`	For a unit clause with no body, replaces two instructions `neck(I,J)` and `foot(N)`. Faster.
`neckcut(I,J)`	Replaces two instructions `neck(I,J)` and `foot(I)`. Faster.
`neckcutfoot(J,N)`	For a clause with just a cut as its body. Replaces three instructions `neck(I,J)`, `cut(I)` and `foot(I)`. Faster.
`fail`	Starts backtracking.

Table A.5: Foot Instructions

Appendix B

Programs

This appendix contains programs and procedures referred to in the main text.

B.1 Assembly Predicates used in the `call/1` built-in predicate

```
assembleGoalSequence(G,I,D,Variant) :-
    var(G),!,
    assemble(G,I,D,Variant).
assembleGoalSequence((G1,G2),I-Id,D-Dd,Variant) :-
    !,
    assemble(G1,I-I1,D-D1,Variant),
    assembleGoalSequence(G2,I1-Id,D1-Dd,Variant).
assembleGoalSequence(G,I,D,Variant) :-
    assemble(G,I,D,Variant).

assemble(V,X,Y,Variant) :-
    var(V),!,
    assemble(call(V),X,Y,Variant).
assemble(V,X,Y,Variant) :-
    integer(V),
    throw(
     error(permission_error,[error_message(['Can''t use an integer as a goal',V])])).
assemble(!,[cut|R]-R,D-D,Variant) :- !.
assemble((X->Y),[expandA3,ifThenCall,landZone|R]-Rd,D-Dd,Variant) :-
    !,
    assembleGoalSequence(X,R-[expandA3,ifThenCommit,word(Variant)|R1],D-D1,0),
    assembleGoalSequence(Y,R1-[expandA3,punctuation|Rd],D1-Dd,Variant).
assemble((X;Y),[expandA3,orCall,landZone,word(0),offset(F),offset(S)|R]-Rd,D-Dd,Variant) :-
```

```
    !,
    assembleGoalSequence(X,R-[expandA3,jump,offset(S),label(F)|R1],D-D1,1),
    assembleGoalSequence(Y,R1-[expandA3,punctuation,label(S)|Rd],D1-Dd,0).
assemble((X,Y),[expandA3,andCall,landZone|R]-Rd,D,Variant) :-
    !,
    assembleGoalSequence((X,Y),R-[expandA3,punctuation|Rd],D,Variant).
assemble(\+X,[expandA3,notCall,landZone,word(0),offset(F)|R]-Rd,D,Variant) :-
    !,
    assembleGoalSequence(X,R-[expandA3,notSucceed,label(F)|Rd],D,Variant).
assemble(X,[expandA3,notCall,landZone,word(1),offset(F)|R]-Rd,D,Variant) :-
    !,
    assembleGoalSequence(X,R-[expandA3,notSucceed,label(F)|Rd],D,Variant).
assemble(not(X),[expandA3,notCall,landZone,word(2),offset(F)|R]-Rd,D,Variant) :-
    !,
    assembleGoalSequence(X,R-[expandA3,notSucceed,label(F)|Rd],D,Variant).
assemble(fail_if(X),[expandA3,notCall,landZone,word(3),offset(F)|R]-Rd,D,Variant) :-
    !,
    assembleGoalSequence(X,R-[expandA3,notSucceed,label(F)|Rd],D,Variant).
assemble(call(X),Y,Z,Variant) :- nonvar(X),!,assemble(X,Y,Z,Variant).
assemble(X,[call,offset(L)|R]-R,[label(L),callFrame,X|D]-D,Variant).

computeOffsets([],[],Lc,Lc).
computeOffsets([label(Lc)|R],S,Lc,Size) :-
    computeOffsets(R,S,Lc,Size).
computeOffsets([offset(L)|R],[word(L-Lc)|S],Lc,Size) :-
    !,
    Lci is Lc+2,
    computeOffsets(R,S,Lci,Size).
computeOffsets([callFrame,X|R],[callFrame,X|S],Lc,Size) :-
    functor(X,_,Arity),
    Lco is Lc+4+2+Arity*4,
    !,
    computeOffsets(R,S,Lco,Size).
computeOffsets([Token|R],[Token|S],Lc,Size) :-
    Lci is Lc+2,
    computeOffsets(R,S,Lci,Size).

resolveOffsets([callFrame,X|R],[X|S]) :-
    !,
    removeCallFrames(R,S).
resolveOffsets([],[]).
resolveOffsets([word(X)|R],[V|S]) :-
    V is X,!,
    resolveOffsets(R,S).
resolveOffsets([T|R],[E|S]) :-
    T equ E,
    resolveOffsets(R,S).

removeCallFrames([],[]).
removeCallFrames([callFrame,X|R],[X|S]) :-
    removeCallFrames(R,S).
```

B.2 The *nrev* Benchmark

This is a complete listing of the file used for the *nrev* benchmark [83] with small modifications to suit Open Prolog.

```
/* BENCH.PL : The classic Prolog benchmark

    Supplied by Quintus Computer Systems, Inc.
    April 30th 1984
*/

/* ======================================================================
   This benchmark gives the raw speed of a Prolog system.

   The measure of logical inferences per second (Lips) used here is taken to
   be procedure calls per second over an example with not very complex
   procedure calls. The example used is that of "naive reversing" a list,
   which is an expensive, and therefore stupid, way of reversing a list.  It
   does, however, produce a lot of procedure calls. (In theoretical terms,
   this algorithm is O(n^2) on the length of the list).

   The use of a single simple benchmark like this cannot, of course, be
   taken to signify a great deal. However, experience has shown that this
   benchmark does provide a very good measure of basic Prolog speed and
   produces figures which match more complex benchmarks. The reason for
   this is that the basic operations performed here: procedure calls with a
   certain amount of data structure access and construction; are absolutely
   fundamental to Prolog execution. If these are done right, then more
   complex benchmarks tend to scale accordingly. This particular benchmark
   has thus been used as a good rule of thumb by Prolog implementors for
   over a decade and forms a part of the unwritten Prolog folklore. So -
   use this benchmark, with this in mind, as a quick, but extremely useful,
   test of Prolog performance.

   In a complete evaluation of a Prolog system you should also be taking
   account speeds of asserting and compiling, tail recursion, memory
   utilisation, compactness of programs, storage management and garbage
   collection, debugging and editing facilities, program checking and help
   facilities, system provided predicates, interfaces to external
   capabilities, documentation and support, amongst other factors.

   ====================================================================== */

/* ----------------------------------------------------------------------
```

```
    get_cpu_time(T) -- T is the current cpu time.

    ** This bit will probably require changes to work on your Prolog
       system, since different systems provide this facility in
       different ways. See your Prolog manual for details.
    ** Also check the code for calculate_lips/4 below.
   ------------------------------------------------------------------- */

get_cpu_time(T) :- statistics(runtime,[T,_]).   /* Quintus Prolog version */

/* get_cpu_time(T) :- T is cputime.        C-Prolog version       */

/* ------------------------------------------------------------------
    nrev(L1,L2)  -- L2 is the list L1 reversed.
    append(L1,L2,L3) -- L1 appended to L2 is L3.
    data(L)      -- L is a thirty element list.

    This is the program executed by the benchmark.
    It is called "naive reverse" because it is a very expensive way
    of reversing a list. Its advantage, for our purposes, is that
    it generates a lot of procedure calls. To reverse a thirty element
    list requires 496 Prolog procedure calls.
   ------------------------------------------------------------------- */

nrev([],[]).
nrev([X|Rest],Ans) :- nrev(Rest,L), append(L,[X],Ans).

append([],L,L).
append([X|L1],L2,[X|L3]) :- append(L1,L2,L3).

data([1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,
              21,22,23,24,25,26,27,28,29,30]).

/* ------------------------------------------------------------------
    lots -- Run benchmark with a variety of iteration counts.

    Call this to run the benchmark with increasing numbers
    of iterations. The figures produced should be about the same -
    except that there may be inaccuracies at low iteration numbers
    if the time these examples take to execute on your machine are
    too small to be very precise (because of the accuracy the
    operating system itself is capable of providing).
    If the time taken for these examples is too long or short then
    you should adjust the eg_count(_) facts.
   ------------------------------------------------------------------- */
```

```
lots :-
%       %add in a garbage collect
%       garbage_collect,
    eg_count(Count),
    bench(Count),
    fail.
lots.

eg_count(10).
eg_count(20).
eg_count(50).
eg_count(100).
eg_count(200).
eg_count(500).
eg_count(1000).
eg_count(2000).
eg_count(5000).
eg_count(10000).

/* ----------------------------------------------------------------------
    bench(Count) -- Run the benchmark for Count iterations.

    bench provides a test harness for running the naive reverse
    benchmark. It is important to factor out the overhead of setting
    the test up and using repeat(_) to iterate the right number of
    times. This is done by running some dummy code as well to see how
    much time the extra operations take.
   ---------------------------------------------------------------------- */

bench(Count) :-
    get_cpu_time(T0),
    dodummy(Count),
    get_cpu_time(T1),
    dobench(Count),
    get_cpu_time(T2),
    report(Count,T0,T1,T2).

/* ----------------------------------------------------------------------
    dobench(Count) -- nrev a 30 element list Count times.
    dodummy(Count) -- Perform the overhead operations Count times.
    repeat(Count)  -- Predicate which succeeds Count times

    This is the supporting code, which is reasonably clear.
   ---------------------------------------------------------------------- */
```

```
dobench(Count) :-
    data(List),
    repeat(Count),
    nrev(List,_),
    fail.
dobench(_).

dodummy(Count) :-
    data(List),
    repeat(Count),
    dummy(List,_),
    fail.
dodummy(_).

dummy(_,_).

repeat(_N).
repeat(N) :- N > 1, N1 is N-1, repeat(N1).

/* -------------------------------------------------------------------
    report(Count,T0,T1,T2) -- Report the results of the benchmark.
    calculate_lips(Count,Time,Lips,Units) --
        Doing Count interations in Time implies Lips lips assuming
        that time is given in Units.

    This calculates the logical inferences per second (lips) figure.
    Remember that it takes 496 procedure calls to naive reverse a
    thirty element list once. Lips, under this benchmark, thus means
    "Prolog procedure calls per second, where the procedure calls
    are not too complex (i.e. those for nrev and append)".

    ** This version of the code assumes that the times (T0.. etc)
       are integers giving the time in milliseconds. This is true for
       Quintus Prolog.  Your Prolog system may use some other
       representation.  If so, you will need to adjust the Lips
       calculation.  There is a C-Prolog version below for the case
       where times are floating point numbers giving the time in
       seconds.
   ------------------------------------------------------------------- */

report(Count,T0,T1,T2) :-
    Time1 is T1-T0,
    Time2 is T2-T1,
    Time  is Time2-Time1,       /* Time spent on nreving lists */
```

```
    calculate_lips(Count,Time,Lips,Units),
    nl,
    write(Lips), write(' lips for '), write(Count),
    write(' iterations taking '), write(Time),
    write(' '), write(Units), write(' ('),
    write(Time2-Time1), write(')'),
    nl.
calculate_lips(_Count,Time,Lips,'msecs') :-     /* Time can be 0 for small */
    Time is 0, !, Lips is 0.        /* values of Count!        */
 /* --- SICStus, Qunitus, etc. version
calculate_lips(Count,Time,Lips,'msecs') :-
    Lips  is (496*float(Count)*1000)/Time.
   --- */

/* --- C-Prolog version

calculate_lips(Count,Time,Lips,'secs') :- Lips  is (496*Count)/Time.

   --- */

%/* --- Open Prolog version

calculate_lips(Count,Time,Lips,'mSecs') :- Lips  is 496*Count/Time.

%   --- *
```

B.3 The *nrev2* code generator

The following code is used to generate and assert a clause containing multiple copies of the same goal, preceded and followed by timing predicates.

```
%for preparing very accurate Lips measurements.
%prepare/3 makes up a clause with N calls to X in it preceded by goal F
%dotiming/0 calls the clause.

%use garbage_collect to reset memory & minimise the possibility of
%garbage collection & shifting.

prepare(F,N,X) :-
        replicateList(F,N,X,Y-[statistics(runtime,[_,T])]),
        formAssertion(timeTrial(T),[F,statistics(runtime,_)|Y],A),
        abolish(timeTrial,1),beep,
```

```
        assert(A),
        write('Please call ''dotiming'' now'),nl.

replicateList(F,N,X,L) :-
        assert('temp$term'(F,X)),
        readNCopies(N,L,F).

readNCopies(0,D-D,F) :- retract('temp$term'(F,_)).
readNCopies(N,[Term|R]-D,F) :-
        N>0,!,
        'temp$term'(F,Term),
        M is N-1,
        readNCopies(M,R-D,F).

formAssertion(Head,TailList,(Head:-Tail)) :-
        formTail(TailList,Tail).

formTail([],true).
formTail([X],X).
formTail([X|Y],(X,R)) :- \+(Y=[]),!,formTail(Y,R).

dotiming :-
        timeTrial(T),
        write('Elapsed time is '),
        write(T),
        write(' mS.'),nl.
```

B.4 Profiling Programs

This section comprises four subsections containing programs related to the generation of profile data for Open Prolog.

B.4.1 The Profiling Code

The following code is used to generate profiles of benchmark programs as specified by the contents of the file 'benchmark stuff'.

```
ensure_loaded(_). %dummy

profile :-
```

```
        open('profile manifest',read,S),
        current_input(I),
        set_input(S),
        read(C),
        process(C),
        set_input(I).

process(end_of_file).
process(bench(F,C,Q)) :-
        name(F,Fs),
        my_append(Fs,".pl",Nfs),
        name(File,Nfs),
        'system$push$display'(message,left,'Now consulting ',File,'',''),
        consult(File),
        'system$set$display'(message,left,F,' being profiled ','',' '),
        run_profile(F,Q,C,-200), %200 microseconds sampling interval
        'system$pop$display'(message),
        abolishAllPredicates,
        consult(profiler),
        !,
        read(N),
        process(N).

abolishAllPredicates :- current_predicate(_,X),functor(X,F,A),abolish(F,A),fail;true.

my_append([],X,X).
my_append([X|Y],Z,[X|A]) :- my_append(Y,Z,A).

%for preparing accurate profiling measurements.

run_profile(Comment,Repeats,Predicate,SamplingInterval) :-
        prepare(Comment,Repeats,Predicate),!,
        garbage_collect,
        'profile$sample$interval'(P,SamplingInterval),
        profileTrial(Ticks,Misses,Nils,Cputime),!,
        name(Comment,Cs),
        my_append(Cs," profile",Rs),
        name(Pf,Rs),
        current_output(Co),
        tell(Pf),
        profile(Ticks,Misses,Nils,Cputime),
        told,
        'profile$sample$interval'(_,P), %reset sampling interval
        set_output(Co).
```

```
%prepare/3 makes up a clause with N calls to X in it,
%doprofile/0 calls the clause.

%use 'abort' to reset memory & minimise the possibility of
%garbage collection & shifting.

prepare(Comment,N,X) :-
replicateList(N,X,Y-['stop$profile','get$profile$tick$count'(T2,M2,N2),!,
  C is cputime-S1,T is T2-T1,M is M2-M1,Ni is N2-N1]),
formAssertion(profileTrial(T,M,Ni,C),['get$profile$tick$count'(T1,M1,N1),
  S1 is cputime,'start$profile'|Y],A),
abolish(profileTrial,4),
abolish('profile run data',3),
assert(A),
asserta('profile run data'(Comment,N,X)).

replicateList(N,X,L) :-
assert('temp$term'(X)),
  readNCopies(N,L).

readNCopies(0,D-D) :- retract('temp$term'(_)).
readNCopies(N,[Term|R]-D) :-
'temp$term'(Term),
M is N-1,
readNCopies(M,R-D).

formAssertion(Head,TailList,(Head:-Tail)) :-
formTail(TailList,Tail).

formTail([],true).
formTail([X],X).
formTail([X|Y],(X,R)) :- formTail(Y,R).

profile(Ticks,Misses,Nils,Cputime) :-
nl,write('Profile Run on '),'system$machine$name'(N),write(N),write(', '),
'system$seconds'(S),'system$date'(S,Date),write(Date),write(' at '),'system$time'(S,Time),
write(Time),nl,
'profile run data'(Comment,Nm,Xm),
write('File is '),write(Comment),nl,
write('Profile of '),
write(Nm),write('iterations of the predicate '),write(Xm),nl,
write('Hit/Miss/Nil Ticks: '),
write(Ticks),write(''),write(Misses),write(''),write(Nils),nl,
interval_in_microseconds(Iv),write(Iv),write('uS per tick.'),
write('Elapsed time: '),
```

```
write(Cputime),write('milliseconds.'),nl,
findall([Name,Usage|Rest],
  ('get$profile$data'(Name,Data),Data=..[Functor,Usage|Rest],Usage>0),
 ProfileMatrix),

%here we have a matrix ordered by rows.
%Empty rows (for code sections with no usages) have been already suppressed. Now
%we need to identify empty columns, i.e. columns with all zeros
%So, construct a list of elements, one per column of the matrix, each of
%which is 'non-zero' if there is at least one non-zero element, and 'zero' otherwise

 non_zero_elements_list(ProfileMatrix,NonZeroList),
 mark_concealed_columns(NonZeroList,ConcealList),
write_concealed_tabbed_list_of_lists([['Name','Uses','Hits','GD1','GD2',
  'G3+','GH','HD1','HD2','HD3+','HH','TA','TG','TL','TH']|ProfileMatrix],ConcealList),
nl.

mark_concealed_columns(
        [N,U,H,G1,G2,G3,G4,H1,H2,H3,H4,T1,T2,T3,T4],
        [N,U,H,G1x,G2x,G3,G4x,H1x,H2x,H3,H4x,T1,T2,T3,T4x]) :-

        T4x is T1, %display trail hits is any trail activity
        H4x is H1+H2+H3,
        H2x is H2+H3,
        H1x is H1+H2+H3,

        G4x is G1+G2+G3,
        G2x is G2+G3,
        G1x is G1+G2+G3.

interval_in_microseconds(I) :-
        'profile$sample$interval'(Iv,Iv),
        (Iv>=0->I is Iv*1000;I is -Iv).

write_concealed_tabbed_list_of_lists([],_).
write_concealed_tabbed_list_of_lists([Line|Rest],ConcealList) :-
         write_concealed_tabbed_line(Line,ConcealList),nl,
         write_concealed_tabbed_list_of_lists(Rest,ConcealList).


write_concealed_tabbed_line([],_).
write_concealed_tabbed_line([X|Y],[0|C]) :-
!,
write_concealed_tabbed_line(Y,C).
write_concealed_tabbed_line([X|Y],[_|C]) :-
```

```
write(X),
put(9),
write_concealed_tabbed_line(Y,C).

non_zero_elements_list(Matrix,[S|R]) :-
        non_zero_column(Matrix,RestOfMatrix,S),!,
        non_zero_elements_list(RestOfMatrix,R).
non_zero_elements_list(_,[]).

non_zero_column([],[],0).
non_zero_column([[0|R]|M],[R|N],X) :-
        !,
        non_zero_column(M,N,X).
non_zero_column([[_|R]|M],[R|N],1) :-
        non_zero_column(M,N,_).
```

B.4.2 The Profile Manifest File

The following is the contents of the file 'profile manifest' specifying the programs upon which the profiles are to be performed. Each term specifies one profile run; for example, the first term specifies that the file 'times10.pl' be consulted, and that 1000 successive calls to the predicate `times10` be profiled.

```
%bench(File,Command,Times).
bench(nreverse,nreverse,1000).

bench(crypt,do_profile,1000).
bench(deriv,do_profile,1000).
bench(poly,do_profile,1000).
bench(primes,do_profile,1000).
bench(qsort,do_profile,1000).
bench(queens,do_profile,100).
bench(query,do_profile,1000).
bench(tak,do_profile,100).
```

B.4.3 Sample Program

The file 'nreverse.pl'—from the Aquarius Benchmarks [79]—is listed here:

```
% generated: 25 October 1989
```

```
% option(s):
%
%   nreverse
%
%   David H. D. Warren
%
%   "naive"-reverse a list of 30 integers

atoms_nreverse :-
        nreverse([a,b,c,d,e,f,g,h,i,j,
        k,l,m,n,o,p,q,r,s,t,u,v,w,x,y,z,a,b,c,d],_).

variables_nreverse :-
        nreverse([A,B,C,D,E,F,G,H,I,J,
        K,L,M,N,O,P,Q,R,S,T,U,V,W,X,Y,Z,A,B,C,D],X).

indeterminate_concatenate :- concatenate(X,[a,b,c],[1,2,3,a,b,c]).

determinate_concatenate :- concatenate([1,2,3],[a,b,c],[1,2,3,a,b,c]).

nreverse :- nreverse([1,2,3,4,5,6,7,8,9,10,11,12,
                      13,14,15,16,17,18,19,20,21,
                      22,23,24,25,26,27,28,29,30],_).

nreverse([X|L0],L) :- nreverse(L0,L1), concatenate(L1,[X],L).
nreverse([],[]).

concatenate([X|L1],L2,[X|L3]) :- concatenate(L1,L2,L3).
concatenate([],L,L).
```

B.4.4 The Boyer Benchmark

```
% generated: 20 November 1989
% option(s):
%
%   boyer
%
%   Evan Tick (from Lisp version by R. P. Gabriel)
%
```

```
%   November 1985
%
%   prove arithmetic theorem

main :- wff(Wff),
rewrite(Wff,NewWff),
tautology(NewWff,[],[]).

wff(implies(and(implies(X,Y),
                and(implies(Y,Z),
                    and(implies(Z,U),
                        implies(U,W)))),
            implies(X,W))) :-
        X = f(plus(plus(a,b),plus(c,zero))),
        Y = f(times(times(a,b),plus(c,d))),
        Z = f(reverse(append(append(a,b),[]))),
        U = equal(plus(a,b),difference(x,y)),
        W = lessp(remainder(a,b),member(a,length(b))).

tautology(Wff) :-
        write('rewriting...'),nl,
        rewrite(Wff,NewWff),
        write('proving...'),nl,
        tautology(NewWff,[],[]).

tautology(Wff,Tlist,Flist) :-
        (truep(Wff,Tlist) -> true
        ;falsep(Wff,Flist) -> fail
        ;Wff = if(If,Then,Else) ->
(truep(If,Tlist) -> tautology(Then,Tlist,Flist)
;falsep(If,Flist) -> tautology(Else,Tlist,Flist)
;tautology(Then,[If|Tlist],Flist),% both must hold
 tautology(Else,Tlist,[If|Flist])
                )
        ),!.

rewrite(Atom,Atom) :-
        atomic(Atom),!.
rewrite(Old,New) :-
        functor(Old,F,N),
```

```
        functor(Mid,F,N),
        rewrite_args(N,Old,Mid),
        ( equal(Mid,Next),        % should be ->, but is compiler smart
          rewrite(Next,New)       % enough to generate cut for -> ?
        ; New=Mid
        ),!.

rewrite_args(0,_,_) :- !.
rewrite_args(N,Old,Mid) :-
        arg(N,Old,OldArg),
        arg(N,Mid,MidArg),
        rewrite(OldArg,MidArg),
        N1 is N-1,
        rewrite_args(N1,Old,Mid).

truep(t,_) :- !.
truep(Wff,Tlist) :- member(Wff,Tlist).

falsep(f,_) :- !.
falsep(Wff,Flist) :- member(Wff,Flist).

member(X,[X|_]) :- !.
member(X,[_|T]) :- member(X,T).


equal(  and(P,Q),
        if(P,if(Q,t,f),f)
        ).
equal(  append(append(X,Y),Z),
        append(X,append(Y,Z))
        ).
equal(  assignment(X,append(A,B)),
        if(assignedp(X,A),
           assignment(X,A),
           assignment(X,B))
        ).
equal(  assume_false(Var,Alist),
        cons(cons(Var,f),Alist)
        ).
equal(  assume_true(Var,Alist),
```

```
        cons(cons(Var,t),Alist)
        ).
equal(  boolean(X),
        or(equal(X,t),equal(X,f))
        ).
equal(  car(gopher(X)),
        if(listp(X),
        car(flatten(X)),
        zero)
        ).
equal(  compile(Form),
        reverse(codegen(optimize(Form),[]))
        ).
equal(  count_list(Z,sort_lp(X,Y)),
        plus(count_list(Z,X),
             count_list(Z,Y))
        ).
equal(  countps_(L,Pred),
        countps_loop(L,Pred,zero)
        ).
equal(  difference(A,B),
        C
        ) :- difference(A,B,C).
equal(  divides(X,Y),
        zerop(remainder(Y,X))
        ).
equal(  dsort(X),
        sort2(X)
        ).
equal(  eqp(X,Y),
        equal(fix(X),fix(Y))
        ).
equal(  equal(A,B),
        C
        ) :- eq(A,B,C).
equal(  even1(X),
        if(zerop(X),t,odd(decr(X)))
        ).
equal(  exec(append(X,Y),Pds,Envrn),
        exec(Y,exec(X,Pds,Envrn),Envrn)
```

```
        ).
equal(  exp(A,B),
        C
        ) :- exp(A,B,C).
equal(  fact_(I),
        fact_loop(I,1)
        ).
equal(  falsify(X),
        falsify1(normalize(X),[])
        ).
equal(  fix(X),
        if(numberp(X),X,zero)
        ).
equal(  flatten(cdr(gopher(X))),
        if(listp(X),
           cdr(flatten(X)),
           cons(zero,[]))
        ).
equal(  gcd(A,B),
        C
        ) :- gcd(A,B,C).
equal(  get(J,set(I,Val,Mem)),
        if(eqp(J,I),Val,get(J,Mem))
        ).
equal(  greatereqp(X,Y),
        not(lessp(X,Y))
        ).
equal(  greatereqpr(X,Y),
        not(lessp(X,Y))
        ).
equal(  greaterp(X,Y),
        lessp(Y,X)
        ).
equal(  if(if(A,B,C),D,E),
        if(A,if(B,D,E),if(C,D,E))
        ).
equal(  iff(X,Y),
        and(implies(X,Y),implies(Y,X))
        ).
equal(  implies(P,Q),
```

```
        if(P,if(Q,t,f),t)
        ).
equal(  last(append(A,B)),
        if(listp(B),
           last(B),
           if(listp(A),
              cons(car(last(A))),
              B))
        ).
equal(  length(A),
        B
        ) :- mylength(A,B).
equal(        lesseqp(X,Y),
        not(lessp(Y,X))
        ).
equal(  lessp(A,B),
        C
        ) :- lessp(A,B,C).
equal(  listp(gopher(X)),
        listp(X)
        ).
equal(  mc_flatten(X,Y),
        append(flatten(X),Y)
        ).
equal(  meaning(A,B),
        C
        ) :- meaning(A,B,C).
equal(  member(A,B),
        C
        ) :- mymember(A,B,C).
equal(  not(P),
        if(P,f,t)
        ).
equal(  nth(A,B),
        C
        ) :- nth(A,B,C).
equal(  numberp(greatest_factor(X,Y)),
        not(and(or(zerop(Y),equal(Y,1)),
                not(numberp(X))))
        ).
```

```
equal(  or(P,Q),
        if(P,t,if(Q,t,f),f)
        ).
equal(  plus(A,B),
        C
        ) :- boyer_plus(A,B,C).
equal(  power_eval(A,B),
        C
        ) :- power_eval(A,B,C).
equal(  prime(X),
        and(not(zerop(X)),
            and(not(equal(X,add1(zero))),
                prime1(X,decr(X))))
        ).
equal(  prime_list(append(X,Y)),
        and(prime_list(X),prime_list(Y))
        ).
equal(  quotient(A,B),
        C
        ) :- quotient(A,B,C).
equal(  remainder(A,B),
        C
        ) :- remainder(A,B,C).
equal(  reverse_(X),
        reverse_loop(X,[])
        ).
equal(  reverse(append(A,B)),
        append(reverse(B),reverse(A))
        ).
equal(  reverse_loop(A,B),
        C
        ) :- reverse_loop(A,B,C).
equal(  samefringe(X,Y),
        equal(flatten(X),flatten(Y))
        ).
equal(  sigma(zero,I),
        quotient(times(I,add1(I)),2)
        ).
equal(  sort2(delete(X,L)),
        delete(X,sort2(L))
```

```
        ).
equal(  tautology_checker(X),
        tautologyp(normalize(X),[])
        ).
equal(  times(A,B),
        C
        ) :- times(A,B,C).
equal(  times_list(append(X,Y)),
        times(times_list(X),times_list(Y))
        ).
equal(  value(normalize(X),A),
        value(X,A)
        ).
equal(  zerop(X),
        or(equal(X,zero),not(numberp(X)))
        ).

difference(X, X, zero) :- !.
difference(plus(X,Y), X, fix(Y)) :- !.
difference(plus(Y,X), X, fix(Y)) :- !.
difference(plus(X,Y), plus(X,Z), difference(Y,Z)) :- !.
difference(plus(B,plus(A,C)), A, plus(B,C)) :- !.
difference(add1(plus(Y,Z)), Z, add1(Y)) :- !.
difference(add1(add1(X)), 2, fix(X)).

eq(plus(A,B), zero, and(zerop(A),zerop(B))) :- !.
eq(plus(A,B), plus(A,C), equal(fix(B),fix(C))) :- !.
eq(zero, difference(X,Y),not(lessp(Y,X))) :- !.
eq(X, difference(X,Y),and(numberp(X),
                          and(or(equal(X,zero),
                                 zerop(Y))))) :- !.
eq(times(X,Y), zero, or(zerop(X),zerop(Y))) :- !.
eq(append(A,B), append(A,C), equal(B,C)) :- !.
eq(flatten(X), cons(Y,[]), and(nlistp(X),equal(X,Y))) :- !.
eq(greatest_factor(X,Y),zero, and(or(zerop(Y),equal(Y,1)),
                                  equal(X,zero))) :- !.
eq(greatest_factor(X,_),1, equal(X,1)) :- !.
eq(Z, times(W,Z), and(numberp(Z),
                      or(equal(Z,zero),
                         equal(W,1)))) :- !.
```

```
eq(X, times(X,Y), or(equal(X,zero),
                     and(numberp(X),equal(Y,1)))) :- !.
eq(times(A,B), 1, and(not(equal(A,zero)),
                      and(not(equal(B,zero)),
                          and(numberp(A),
                              and(numberp(B),
                                  and(equal(decr(A),zero),
                                      equal(decr(B),zero)))))))) :- !.
eq(difference(X,Y), difference(Z,Y),if(lessp(X,Y),
                                       not(lessp(Y,Z)),
                                       if(lessp(Z,Y),
                                          not(lessp(Y,X)),
                                          equal(fix(X),fix(Z))))) :- !.
eq(lessp(X,Y), Z, if(lessp(X,Y),
                     equal(t,Z),
                     equal(f,Z))).

exp(I, plus(J,K), times(exp(I,J),exp(I,K))) :- !.
exp(I, times(J,K), exp(exp(I,J),K)).

gcd(X, Y, gcd(Y,X)) :- !.
gcd(times(X,Z), times(Y,Z), times(Z,gcd(X,Y))).

mylength(reverse(X),length(X)).
mylength(cons(_,cons(_,cons(_,cons(_,cons(_,cons(_,X7)))))),
         plus(6,length(X7))).

lessp(remainder(_,Y), Y, not(zerop(Y))) :- !.
lessp(quotient(I,J), I, and(not(zerop(I)),
                            or(zerop(J),
                               not(equal(J,1))))) :- !.
lessp(remainder(X,Y), X, and(not(zerop(Y)),
                             and(not(zerop(X)),
                                 not(lessp(X,Y))))) :- !.
lessp(plus(X,Y), plus(X,Z), lessp(Y,Z)) :- !.
lessp(times(X,Z), times(Y,Z), and(not(zerop(Z)),
                                  lessp(X,Y))) :- !.
lessp(Y, plus(X,Y), not(zerop(X))) :- !.
lessp(length(delete(X,L)), length(L), member(X,L)).
```

```
meaning(plus_tree(append(X,Y)),A,
        plus(meaning(plus_tree(X),A),
             meaning(plus_tree(Y),A))) :- !.
meaning(plus_tree(plus_fringe(X)),A,
        fix(meaning(X,A))) :- !.
meaning(plus_tree(delete(X,Y)),A,
        if(member(X,Y),
           difference(meaning(plus_tree(Y),A),
                      meaning(X,A)),
           meaning(plus_tree(Y),A))).

mymember(X,append(A,B),or(member(X,A),member(X,B))) :- !.
mymember(X,reverse(Y),member(X,Y)) :- !.
mymember(A,intersect(B,C),and(member(A,B),member(A,C))).

nth(zero,_,zero).
nth([],I,if(zerop(I),[],zero)).
nth(append(A,B),I,append(nth(A,I),nth(B,difference(I,length(A))))).

boyer_plus(plus(X,Y),Z,
     plus(X,plus(Y,Z))) :- !.
boyer_plus(remainder(X,Y),
     times(Y,quotient(X,Y)),
     fix(X)) :- !.
boyer_plus(X,add1(Y),
     if(numberp(Y),
        add1(plus(X,Y)),
        add1(X))).

power_eval(big_plus1(L,I,Base),Base,
           plus(power_eval(L,Base),I)) :- !.
power_eval(power_rep(I,Base),Base,
           fix(I)) :- !.
power_eval(big_plus(X,Y,I,Base),Base,
           plus(I,plus(power_eval(X,Base),
                       power_eval(Y,Base)))) :- !.
power_eval(big_plus(power_rep(I,Base),
                    power_rep(J,Base),
                    zero,
                    Base),
```

```
          Base,
          plus(I,J)).

quotient(plus(X,plus(X,Y)),2,plus(X,quotient(Y,2))).
quotient(times(Y,X),Y,if(zerop(Y),zero,fix(X))).

remainder(_,         1,zero) :- !.
remainder(X,         X,zero) :- !.
remainder(times(_,Z),Z,zero) :- !.
remainder(times(Y,_),Y,zero).

reverse_loop(X,Y,  append(reverse(X),Y)) :- !.
reverse_loop(X,[], reverse(X)          ).

times(X,         plus(Y,Z),      plus(times(X,Y),times(X,Z))      ) :- !.
times(times(X,Y),Z,              times(X,times(Y,Z))              ) :- !.
times(X,         difference(C,W),difference(times(C,X),times(W,X))) :- !.
times(X,         add1(Y),        if(numberp(Y),
                                    plus(X,times(X,Y)),
                                    fix(X))                       ).
```

Appendix C

Miscellaneous

Figure C.1: The Splash Screen of Tricia Prolog, listing the authors and contributors.