xpProlog: High Performance Extended Pure Prolog

by

PETER GERALD LÜDEMANN

B. Sc., The University of British Columbia, 1975

A THESIS SUBMITTED IN PARTIAL FULFILLMENT

OF THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE

in

THE FACULTY OF GRADUATE STUDIES

DEPARTMENT OF COMPUTER SCIENCE

We accept this thesis as conforming
to the required standard

THE UNIVERSITY OF BRITISH COLUMBIA

February 1988

3 9

Abstract

Adhering to the principles of logic programming results in greater expressiveness than is obtained by using the many non-logical features which have been grafted onto current logic programming languages such as Prolog. This report describes an alternative approach to high performance logic programming in which the language and its implementation were designed together. Prolog's non-logical features are discarded and new logical ones are added. *Extended pure Prolog* (xpProlog) is a superset of conventional Prolog; it is sufficient in itself, without any need for "impure" non-logical predicates. This gives both greater expressiveness and better performance than conventional Prologs.

XpProlog programs have the following advantages over conventional Prolog programs:

- They are often easier to understand because their meaning does not rely on the underlying computational mechanism.

- Coroutining, automatic delaying and sound negation are available.
- As technology improves, better implementations and optimization techniques can be used without affecting existing programs.

This report covers:

- The proper use of logic programming.
- How Prolog must be changed to become a good logic programming language (**xpProlog**).
- Sound negation and coroutining.
- An efficient abstract machine (**xpPAM**) which can be efficiently emulated on conventional machines, translated to conventional machine code, or implemented in special purpose hardware.
- How to compile extended Prolog and functional (applicative) languages to the abstract machine or to conventional machine code.
- Discussion of alternative Prolog abstract machine designs.

The **xpProlog** Abstract Machine's design allows:

- Performance similar to the Warren Abstract Machine (WAM) for sequential programs.
- Tail recursion optimization (TRO).
- Parallelism and coroutining with full backtracking.
- Dynamic optimization of clause order.
- Efficient *if-then-else* ("shallow" backtracking).
- Simple, regular instruction set for easily optimized compilation.
- Efficient memory utilization.

- Integrated object-oriented virtual memory.
- Predicates as first-class objects.
- Simple extension to functional programming.

C.R. categories: **I.2.5**: Prolog; **D.1.3**: concurrent programming; **D.3.2**: very high level languages; **D.3.3**: language constructs: coroutines, backtracking; **D.3.4**: interpreters.; **I.2.3**: logic programming.

Contents

Preface

He had explained this to Pooh and Christopher Robin once before, and had been waiting ever since for a chance to do it again, because it is a thing which you can easily explain twice before anybody knows what you are talking about.
— A. A. Milne, *Winnie the Pooh*

Je n'ai fait celle-ci plus longue que parce que je n'ai pas eu le loisir de la faire plus courte.
(I have made this letter longer than usual, only because I have not had the time to make it shorter.)
— Blaise Pascal, *Lettres Provinciales (1656 – 1657)*

This report is the result of several year's investigation into logic programming and its most popular realization: Prolog. I started by wanting to design a typesetting system, implemented in Prolog but the Prologs that were available then were too slow and lacked some necessary features such as coroutining and sound negation. I therefore decided to investigate logic programming and implement an interpreter.

Implementing a compiler and interpreter is not a task to be undertaken lightly. I have worked part time on this for three years and I have not produced a production quality implementation. However, I have succeeded in demonstrating the possibilities of logic programming. Three or four people should be able to produce a production quality implementation of this in about a year.

... when you are a Bear of Very Little Brain, and you think of things, you find that sometimes that a Thing which seemed very Thingish inside you is quite different when it gets into the open and has other people looking at it.

— A. A. Milne, *Winnie the Pooh*

This report contains two themes: pure logic programming and an abstract machine for implementing pure logic programming. These themes affect each other because I want to show that pure logic programming can have an efficient implementation. I wish to put forward my abstract machine design as a viable alternative to the popular Warren Abstract Machine (WAM) [Warren 1983], so I have described the implementation first and the language second.

This work describes practical issues which have been sufficiently implemented to give an existence proof of their efficacy. Some ideas in this report have come from analysis of the implementation and have not yet found their way into the implementation. Most of the abstract machine ideas have been implemented but some of the language ideas have not.

There are no benchmark data in this report. The usual benchmark (naïve reverse) is probably not a very good predictor of over-all execution speed. My implementation has achieved competitive speed for this simple benchmark, compared against published figures for the fastest commercial Prolog implementations. However, proper speed evaluation would take many months of hard work and is beyond the scope of this report.[1]

Acknowledgments

To spend too much time in studies is sloth.
— Francis Bacon, *Of Studies*

This research has been partially funded by a Shared University Grant from IBM Canada Ltd.

I thank Harvey Abramson and Douglas Teeple for suggesting that I become involved in logic programming (Harvey also let me use his extensive collection of papers and provided the introductory material for the chapter on compiling a functional language); Block Bros. Data Centre which let allowed me the

[1] The interested reader might obtain the ECRC benchmarks, published on *Usenet*. Other benchmarks exist, for example, Evan Tick's translation of the LISP benchmarks described in [Gabriel 1985].

necessary flexibility in my work schedule; Lee Naish, Tom Rushworth, Paul Voda and Marc Gillet for listening, criticizing and suggesting; Romas Aleliunas for extensive editing suggestions; and my wife, Kikuyo, for letting me leave many household chores undone while I worked on this project.

Pure logic programming

Art, it seems to me, should simplify. That, indeed, is very nearly the whole of the higher artistic process; finding what conventions of form and what detail one can do without and yet preserve the spirit of the whole — so that all that one has suppressed and cut away is there to the reader's consciousness as much as if it were in type on the page.
— Willa Cather, *On the Art of Fiction*

1.1 What is wrong with impure logic programming

A typical attitude to non-logical features in logic programming is [Cohen and Feigenbaum 1982, p. 123]:

> To a certain extent, the development of logic programming has followed the pattern of LISP. Both languages are founded on clear, mathematically

motivated foundations. Both languages have a side-effect-free kernel and a procedural interpretation that can be defined in a simple and elegant fashion. Yet both language families have yielded to the practical needs of their user communities and have incorporated numerous features that detract from their underlying elegance in favor of improved convenience and efficiency. In a sense, the fact that logic programming has progressed to the point of incorporating such features attests to its practicality and growing popularity.

This author admits that non-logical features are wrong, yet he defends them because of their "improved convenience and efficiency." This brings to mind the "*go to* considered harmful" [Dijkstra 1968] controversy. Many letters were written by people, claiming that *go to*s were needed because of their convenience and efficiency. We now know that programs written in a disciplined style, without *go tos*, are usually clearer and just as efficient (if not more so) than *go to*-filled spaghetti code. Similar results are now being published for *cut* [Debray 1986] [O'Keefe 1985].

This report describes the implementation of a pure logic programming language. The language's features remove any excuses for writing bad, non-logical programs. The vision expressed in "Algorithm = Logic + Control" [Kowalski 1979b] is possible. And programs written with such a pure logic programming language will run as fast as programs written in conventional Prolog – sometimes faster.

1.2 On good design of a logic programming language

Simplicity requires much thought; complexity merely requires much work. But many computer language and machine designers refuse to see the advantages of simplicity – they correct perceived defects by adding new "features," thereby burying a simple design under a mass of graceless and cumbersome accretions. Eventually, the whole ungainly mess must be replaced by something new.

A programmer's major tool is her programming language. She uses it every day; its quality continually affects her. No programming language can stop a bad programmer from writing bad programs; but a bad programming language can prevent a good programmer from producing good programs.

Logic programming is a simple idea which is already in danger of being buried under a myriad of new "illogical" features. Prolog is already encumbered this way – the result is less powerful than a simple design based strictly on the original concepts.

Programming language designers must strive to give the highest quality product possible, with every part carefully considered as to how it will help good programmers to produce good programs. The language designer must never succumb to doing "what is 'reasonable' even when it isn't any good" [Pirsig 1974] – his motto should be: "Only the good; never the expedient."

A simple design is not necessarily easy to implement. The small number of basic principles must fit together well without duplication. Any careless side effects will prevent some pieces fitting together. The designer needs much discipline to not deviate from the basic principles by introducing special cases and restrictions. This discipline results in a product which is easy to use and easy to extend.

1.3 Principles of logic programming and pure Prolog

Louis XVI: *C'est une grande révolte.*
Le Rochefoucauld-Liancourt: *Non, Sire, c'est une grande révolution.*

The central ideas of logic programming are:

- The most interesting thing about a program is what it does, not the exact sequence of computations.
- The best way of describing a program is by writing specifications using a subset of mathematical logic.
- When the logical specifications are stated using Horn clauses, they can be executed efficiently.
- There is no concept of time or sequence in the meaning of a logic program (execution may, of course, be sequential).

The programmer does not need to write a program, only the specifications. This is "declarative programming." Logic programming offers a solution to one of the problems of software engineering: precise problem specification. Control information may be added to a logic program, as suggested by *"Algorithm = Logic + Control"* [Kowalski 1979b]. This additional control information does not affect a program's meaning; it just improves execution speed.

Logic programming is new to many computer scientists and programmers. A new way of thinking is required to write good logic programs. Without a new way of thinking, programmers produce programs which are full of non-logical features such as *cut* ("!") or *var* – these are really disguised conventional programs which have been hacked to look like logic programs. These non-logical features are not necessary. See [Walker, McCord, Sowa and Wilson 1987, Chapter 3] for methods of avoiding or localizing non-logical features.

Early implementers of Prolog "solved"its space-and time-inefficiencies, by adding poorly conceived "non-logical" features. For a discussion of non-logical constructs and their inconsistent implementations, see [Moss 1986]. Instead of patching existing implementations, *extended pure Prolog* (xpProlog) generalizes the original concept. Syntactically, it is similar to Prolog, but:

- *Extended*: with control rules which add flexibility to Prolog's strict left-to-right depth-first computation rule, allowing natural specification of a larger set of programs than ordinary Prolog. Control can be specified separately from predicate definitions.
- *Pure*: without (and not needing) non-logical predicates such as *cut* ("!"), *var*, etc..

Although xpProlog's syntax is similar to Prolog's syntax, some xpProlog programs will not work on a conventional Prolog implementation – they will either go into infinite loops or run very slowly. XpProlog programs can be given an obvious declarative reading in first-order logic – this is not true of many conventional Prolog programs. XpProlog programs' behaviour do not depend on execution order, although efficiency will depend on execution order. XpProlog's greater expressive power costs almost nothing in efficiency; sometimes it greatly increases efficiency.

Prolog is based on the Horn clause subset of first order logic. Negation, first and second order logic and set theory can be added [Lloyd and Topor 1984] [Voda 1986]. These are implemented unsoundly in conventional Prolog, using non-logical predicates such as *cut* ("!") and *var*. Sound implementations are no more difficult than unsound implementations, as will be shown.

Warren's pioneering design for implementing logic programming (the "Warren Abstract Machine" or WAM) [Warren 1977] [Warren 1983] [Gabriel, Linkholm, Lusk and Overbeek 1985] proves that logic programming languages can be executed as efficiently as other symbol-oriented languages (see [Tick 1986] for a comparison with LISP). WAM allows efficient execution on both conventional and special purpose hardware (for example [Dobry, Patt and Despain 1984] [Tick and Warren 1984] [Dobry 1987]). The **xpProlog** Abstract Machine (**xpPAM**) is a significant modification of WAM, to support the flexible execution order needed for coroutining and sound negation. Prolog code can be compiled to **xpPAM** more easily than to WAM; execution performance is similar for both machines.

This paper, then, explores some extensions to Prolog – *Extended pure Prolog* – which remove restrictions from conventional designs. **XpProlog** often allows more compact, understandable and efficient programs than conventional Prolog allows.

I will start by discussing the implementation `xpProlog` and its abstract machine `xpPAM`. The abstract machine can support conventional Prolog, `xpProlog` and functional languages – I will focus mainly on its support of `xpProlog`. But the reader should be careful do distinguish between what is imposed by the abstract machine, what is imposed by the compiler and what is part of the language design.

Efficient implementation of `xpProlog`, like efficient implementation of conventional Prolog, requires a good compiler. The techniques for such a compiler will also be discussed.

1.4 Background

If I have seen further it is by standing on the shoulders of giants.
— Sir Isaac Newton, *Letter to Robert Hooke*

I assume that the reader has some basic knowledge of mathematical logic, logic programming and conventional Prolog. [Kowalski 1979] is probably the best place to start because he concentrates on logic and avoids being constrained to one particular implementation. Texts on conventional Prolog include [Walker, McCord, Sowa and Wilson 1987], [Sterling and Shapiro 1986], [Bratko 1986], [Kluzniak and Szpowiez 1985], [Clark and McCabe 1984] and [Clocksin and Mellish 1981]. Mathematical logic is covered in [Quine 1941], [Kleene 1967] and [Hodges 1977].

A glossary is provided at the end of this report.

I do not assume any knowledge of the Warren Abstract Machine (WAM). The papers which describe the WAM often gloss over some of the more subtle design details which contribute crucially to its speed. Many of these details also exist in my design and I will try to explain them fully.

My ideas about the control rules for unrestricted pure Prolog derive from [Naish 1985b]. The control rules are sound [Lloyd 1984, pp. 45-47]. The implementation techniques derive from the Warren Abstract Machine [Warren 1983]. I have made some small changes and extensions to Naish's ideas and significant changes to Warren's. I recommend reading [Naish 1985b] although I will present many of his ideas, but in a slightly different form.

1.5 Delay notation

`XpProlog` is pure Prolog, extended for delays. In `xpProlog`, any variable may be suffixed by a question mark ("?"). This will cause the predicate to delay until the variable becomes instantiated. This instantiation is required only at the "top level"; for example, if the variable became instantiated to a list element ("cons cell"), the head and tail would not necessarily need to be instantiated for execution to resume.

The "?" is propagated outward from a compound term. "`X?.Y`" is the same as "`(X?.Y)?`" – the parameter must be instantiated to a list element, the head must also be instantiated, but the tail of the list element need not be instantiated. `XpProlog`'s "?" differs from the "?" in Concurrent Prolog [Shapiro 1983]. For example:

```
pred(X?.Rest, Y) :- test1(X), test2(Y?), test3(X, Y).
```

will delay until the first parameter is instantiated to a list element with its head (X) also instantiated. Once these have become instantiated, `test1` will be tried. If `test1` succeeds, `pred` will delay until `Y` becomes instantiated, after which `test2` and `test3` are tried.

A delayed predicate acts as if it had succeeded. When the required variables become instantiated, the delayed predicate's execution is resumed. Using the above example of `pred` with the query

```
?- pred(A, B), A = (H.T), H = a, Y = b.
```

execution will proceed:

`pred(A, B)`	`pred` delays on `A`
`A = (H.T)`	instantiates `A`
`pred(H.T, B)`	`pred` resumes and delays on `H`
`H = a`	instantiates `H`
`pred(a.T, B)`	`pred` resumes
`test1(a)`	(succeeds)
`test2(B)`	`pred` delays on `B`; `test2` is **not** tried
`Y = b`	instantiates `Y`
`test2(b)`	`pred` resumes; tries `test` (succeeds)
`test3(a, b)`	(succeeds)

Instead of marking predicates with "?"s, separate *proceed* declarations may be used. Thus,[2]

2 Here, the "x," "`rest`" and "`result`" are comments. "`Proceed`" acts like an ordinary predicate which tries to match question marks. It is an error to call `proceed` with any

```
?- proceed predx(x?.rest, result). /* "x", "rest", "result" are comments */
predx([], []).
predx(X.Rest, Result) :- ...
```

is the same as

```
predx([]?, []).
predx(X?.Rest, Result) :- ...
```

The main advantage of *proceed* declarations is that they are separate from the clauses. They can also be used when a predicate must delay until any one of several variables become instantiated. Multiple *proceed* declarations are or-ed. An example of this is *append* when used to implement *append3* for combining (or splitting) three lists:

```
?- proceed append(a?, b, c).
?- proceed append(a, b, c?).
append([], X, X).
append(X.A, B, X.C) :- append(A, B, C).
append3(A, B, C, D) :- append(A, B, Z),
                       append(Z, C, D).
```

Append will proceed (not delay) only if the first argument or if the third argument is instantiated; otherwise it will delay. *Proceed* declarations are not needed for *append3* because all the delaying is done within *append.*

Without the *proceed* declarations, this would go into an infinite loop for the query

uninstantiated variables because these will incorrectly unify with question marks. The declaration could be given `?-proceed predx(?.-, -).`

```
?- append3(1.W, X, Y, 2.Z).
```

and there is no way to re-order the clauses of *append* to prevent the infinite loop.

The effect of the *proceed* declarations for *append* is given by the following pseudo-code (which is **not** supported by xpProlog):

```
append(P1, P2, P3) :-
    ifvar P1 then
        ifvar P3 then orDelay(P1, P3)
        else a(P1, P2, P3)
    else a(P1, P2, P3).

a([], X, X).
a(X.A, B, X.C) :- append(A, B, C). /* Note: calls "append", not "a" */
```

The pseudo-control structure *ifvar* tests the variable for being instantiated. The pseudo-control predicate *delay* suspends the predicate until any one of its arguments becomes instantiated – execution then resumes at the beginning of the predicate. The *ifvar* pseudo-control structure is similar to conventional Prolog's *var* predicate. It can be used correctly only if it is used only with other *var* predicates or with *delay*. XpProlog's *proceed* declarations are more readable and safer than *ifvar* and *delay*, so only *proceed* declarations are provided in the language.

Examples of using delays are given in sections 9.2, "Coroutining example" on page 187, section 9.3, "Test and generate" on page 191 and section 9.4, "Pseudo-parallelism" on page 194.

1.6 Organization of the Report

The rest of this report consists of two sections:

- the abstract machine (`xpPAM`) and its implementation
- the extended Prolog language **`xpProlog`** and examples of its use.

The two sections are almost independent of each other. The common theme is delaying which allows predicates to be tried in a different order from conventional Prolog's strict left-to-right top-down order. The "?" operator and *proceed* declarations (which mark delays) are explained in section 1.5, "Delay notation" on page 8.

The abstract machine

The extended Prolog abstract machine (xpPAM) can be used to implement conventional Prolog, extended pure Prolog (xpProlog) or pure functional programming languages. This report will concentrate on xpProlog implementation.

The extended Prolog language (xpProlog) is similar to conventional Prolog, except for:

- No "impure" non-logical predicates.
- Delay notation (described in section 1.5, "Delay notation" on page 8).
- More control structures such as *if-then-else*.
- Meta-variables for all-solutions predicates.

XpPAM has superficial similarities with the Warren Abstract Machine (WAM) [Warren 1983]. Programs can run on xpPAM about as quickly as they can run on WAM. However, xpPAM is much simpler than WAM.

XpPAM can be thought of either as the design for a logic programming engine (implemented as an interpreter or in hardware) or as an intermediate code for producing machine code on conventional hardware.

The description is divided into sections:

- Section 2.0, "Fast append on a conventional machine" on page 15 uses the *append* predicate as an example of how Prolog can be compiled to very efficient machine code on a conventional machine using C and IBM/370 assembler.
- Section 3.0, "The basic sequential inference engine" on page 28 describes the basic machine and lists all its instructions, skipping over the non-deterministic features.
- Section 4.0, "Backtracking and delaying" on page 66 describes how the basic sequential engine is extended to allow backtracking and delaying.
- Section 5.0, "Compiling Prolog" on page 85 describes a compiler for producing good abstract machine code.
- Section 6.0, "Compiling a functional language to the logic engine" on page 116 describes how functional languages can be compiled to efficient xpPAM code.
- Section 7.0, "Design decisions" on page 138 discusses some of the design trade-offs in xpPAM.

2.0 Fast append on a conventional machine

You must lie upon the daisies and discourse in novel phrases of your complicated state of mind,
The meaning doesn't matter if it's only idle chatter of a transcendental kind.
And everyone will say,
As you walk your mystic way,
"If this young man expresses himself in terms too deep for me,
Why, what a very singularly deep young man this deep young man must be!"
— Sir William S. Gilbert, *Patience, act I*

I do loathe explanations.
— J. M. Barrie, *My Lady Nicotine*

To illustrate what a Prolog implementation should do on a conventional machine, I will translate deterministic *append* into C. The xpPAM abstract machine and backtracking will be introduced in a later chapter.

```
append([], X, X).
append(X.A, B, X.C) :- append(A, B, C).
```

A good Prolog optimizing compiler should detect common patterns of code and translate them into special sequences. The *append* predicate is typical of a more general sequence:

```
p([], []). /* terminate at nil */
p(A.X, A2.X2) :- q(A, A2), /* perform some operation on A, giving A2 */
                 p(X, X2). /* continue with the rest of the list */
```

This is often deterministic and can therefore be handled well by conventional machines. On an IBM/370, the *append* inner loop can be reduced to 8 machine instructions, compared to 2 instructions for **xpPAM** abstract machine (7 for WAM[3]). For mostly deterministic predicates, a conventional machine can therefore be as fast as special purpose hardware.

3 The **xpPAM** code is given later in this chapter. The WAM code is:

```
append/3: switchonterm ... []=>L1, list=>L2, var=>L0
L1:       get_nil   1
          get_value 2,3
          proceed
L2:       getlist   1
          unify_var 4
          unify_var 1
          getlist   3
          unify_val 4
          unify_var 3
          execute   append/3
L0:       try       L1
          trust     L2
```

This has 8 instructions in the inner loop. It can be reduced to 7 by replacing the last `execute` by the first `switchonterm`.

I will describe the Prolog equivalent of the LISP function

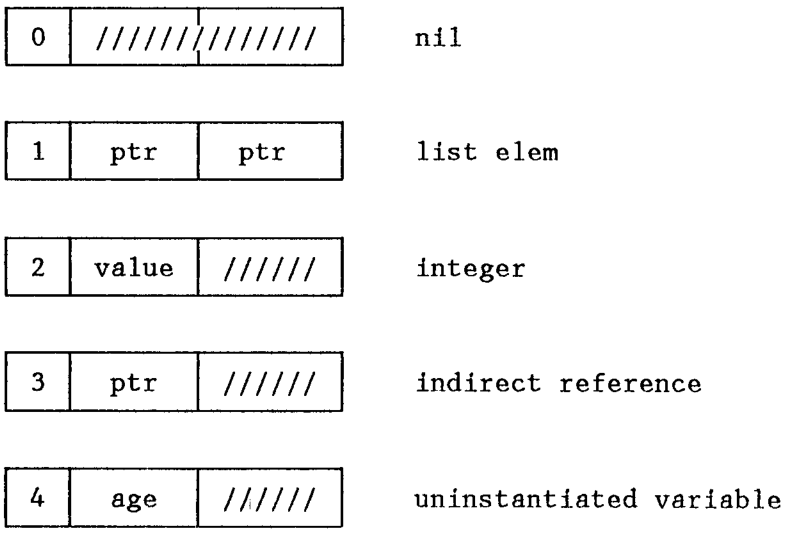
```
(defun append(A B)
   (cond ((nil A) B)
         (T      (cons (car A) (append (cdr A) B)))))
```

However, there is an important difference: the Prolog version is tail-recursive but the LISP version is not (the recursive call to LISP's *append* is inside a call to *cons*). To produce the iterative code given below, the LISP code requires a more complex transformation than does the Prolog code.

2.1 Data structures

First, the data types to define a value cell (slightly simplified from the actual implementation):[4]

4 Many implementations save space by using variable size value cells and by coding information directly when possible. Variable size cells do not change the accessing code but they do complicate the heap manager. However, more savings can be had by avoiding pointers. For example, the list `[1,2]` could be fit into one list element if a list element contained two value cells instead of two pointers: the values 1 and 2 could be of type "short integer" which fits inside a word. Such a scheme complicates accessing elements; I have chosen implementation simplicity and speed over memory usage (although this scheme could be worked into my design). Other list compaction schemes such as "cdr-coding" could also be used but they also complicate the implementation, particularly unification.


0 ////////////// nil
1 ptr ptr list elem
2 value ////// integer
3 ptr ////// indirect reference
4 age ////// uninstantiated variable

In C, this would be expressed:

```
/* value cell tags: */
enum tgE = {tgNil=0, tgListElem=1, tgInteger=2, tgReference=3,
            tgVariable=4,  /* uninstantiated variable */
            tgUnalloc=5 }; /* unallocated: on free list */

union cellUnion {
    int            asInteger;
    int            varAge;  /* "age" of the variable */
    struct cell   *asReference;
    struct { struct cell *head, *tail; } asListElem;
};
struct cell {
    enum tgE tag; /* tgNil, tgListElem, ..., tgVariable */
    union cellUnion u;
};
```

The contents of a value cell depends on the `tag`. A list is made up of list elements (tag `tgListElem`) each of which contains a head and a tail. The last element is *nil* (tag `tgNil`). A "reference" cell contains a pointer to another value cell – it is invisible to the user because it is always dereferenced whenever it is used. An uninstantiated variable (tag `tgVariable`) is a cell which has not yet been assigned a value. When the value is determined, the tag is changed and the new value filled in. The "age" of a variable is used to determine whether or not information must be saved when the variable becomes instantiated, to allow backtracking (this corresponds to the Warren Abstract Machine method of determining the variable's age by a comparing stack positions).

2.2 C code for append function

Here is the most efficient version of *append* possible with the above value cell definitions. It assumes that there are no "reference" cells and that the third argument is always the output (that is, *append* is not used to split a list) – these extra complications will be dealt with later.

The *append* function takes two value cell pointers (*p1, p2*) and a pointer to a value cell pointer (*p3*) to return the new list (in a language like Pascal, *p3* would be a *var* parameter; C requires declaring it as a pointer). The algorithm is:

```
while p1 points at a list element cell {
    allocate a new list element cell; assign it to what p3 points at
    make p1 point to the next input list cell (or nil if at end of list)
    make p3 point to the tail of the new list cell
}
if p1 points at nil, assign p2 to what p3 points at
     otherwise raise error condition
```

Note that for speed, the tail of each new list element is not filled in; however, *p3* points at it and the next iteration fills it in. Here is the C code. The *allocListElem(Head,Tail)* procedure is used to allocate new list cells.[5] The *dummyCell* is strictly a place holder for *allocListElem*; its contents do not matter at all.

```
void append(p1, p2, p3)
struct cell *p1, *p2, **p3; /* p3: var cellPtr */
{ while (p1->tag == tgListElem) {
      *p3 = allocListElem(p1->u.asListElem.head, &dummyCell);
      p1 = p1->u.asListElem.tail;
      p3 = &((*p3)->u.asListElem.tail);
  }
  if (p1->tag == tgNil) { *p3 = p2; }
                   else { error(); }
}
```

The query

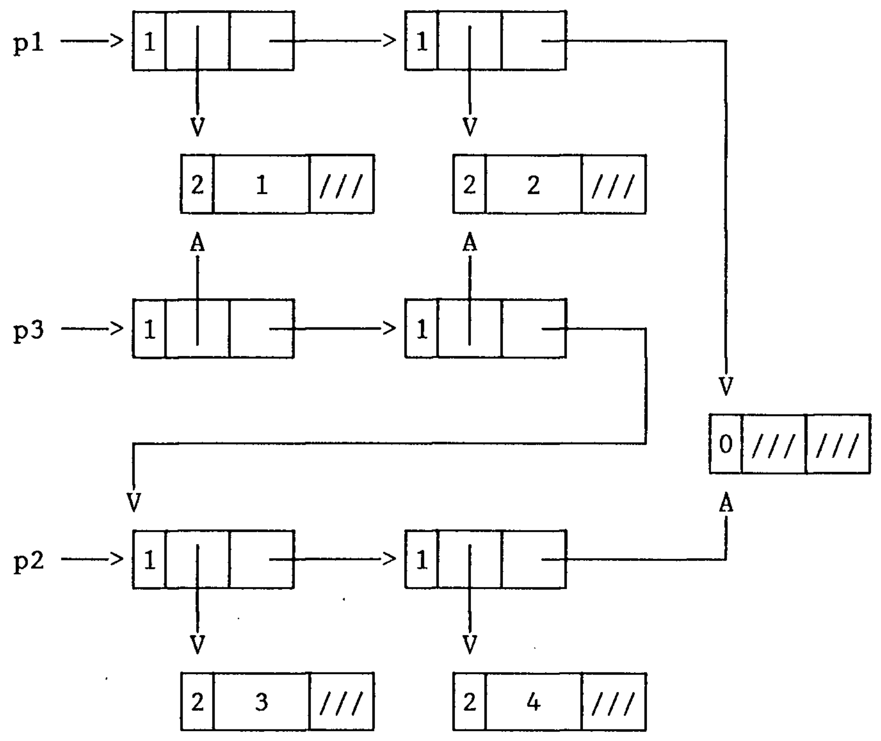
```
?- append([1,2], [3,4], L3).
```

is coded:

```
  struct cell *p1, *p2, *p3;
  p1 = allocListElem(allocInteger(1),
       allocListElem(allocInteger(2), &nilCell)));
  p2 = allocListElem(allocInteger(3),
       allocListElem(allocInteger(4), &nilCell));
  /* p3 does not need to be initialized */
  append(p1, p2, &p3);
```

and results in (note that *p3* is shown between *p1* and *p2*):

5 In the actual program, this statement is written slightly differently, using a macro.



Optimization to this level is only possible if the compiler knows that there are no "reference" cells. If there are reference cells, they must be dereferenced each time they are used by:

```
while (p1->tag == tgReference) { p1 = p1->u.asReference; }
```

This will never go into an infinite loop because the unification algorithm guarantees that a reference cell will never point to itself.

2.3 C code for deterministic append predicate

The above C code in effect implements a function which returns a value in *p3*. If *p3* is allowed to be any kind of value, the following code will do the job. For brevity, only code for *p1* as (a reference to) *nil* or a list cell and for *p3* as a reference cell or uninstantiated variable is shown; the other cases generate calls to *error*. The full implementation has code for handling backtracking when *p1* is an uninstantiated variable — this is discussed in section 4.0, "Backtracking and delaying" on page 66.

```
void append(p1, p2, p3) /* append using tgVariable, tgReference */
struct cell *p1, *p2, *p3;
{
start:
  switch (p1->tag) {
  case tgReference:
    p1 = p1->u.asReference;
    goto start;
  case tgNil:
    while (p3->tag == tgReference)
        { p3 = p3->u.asReference; } /* deref p3 */
    if (p3->tag == tgVariable) {
      pushOnResetStack(p3);
      p3->tag = tgReference;
      p3->u.asReference = p2;
    } else { error(); }
    break;
  case tgListElem:
    while (p3->tag == tgReference)
        { p3 = p3->u.asReference; } /* deref p3 */
    if (p3->tag == tgVariable) {
      pushOnResetStack(p3);
      p3->tag = tgReference;
      p3->u.asReference = allocListElem(p1->u.asListElem.head,
                                        allocVariable());
      /* tail recursive call... */
      p1 = p1->u.asListElem.tail;
      p3 = p3->u.asReference->u.asListElem.tail;
      goto start; /* tail recursion optimisation */
    } else { error(); }
    break;
  default:
    error();
    }
}
```

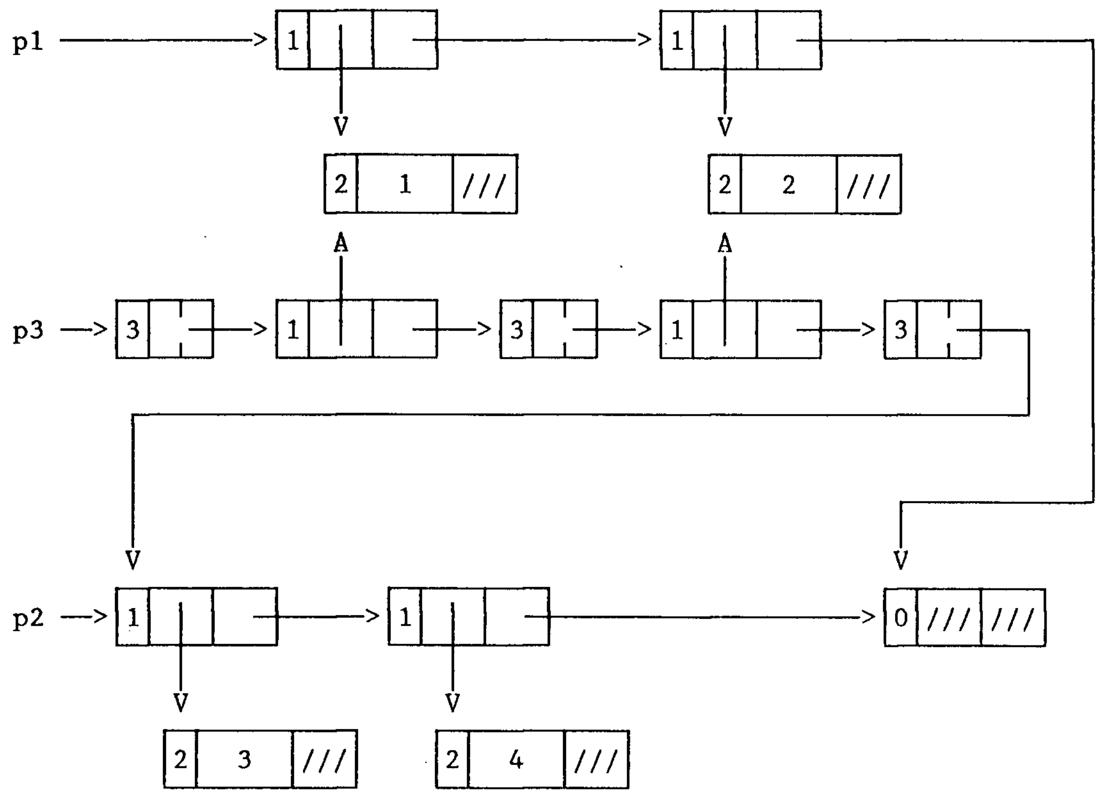
Explanation. The first case of *append* dereferences *p1* by guaranteeing that it is not a reference cell. Once *p1* is fully dereferenced, one of the other cases is executed. If *p1* is *nil*, *p2* and *p3* are unified (the above code does only the situation where *p3* is an uninstantiated variable). If *p1* is a list element, *p3* is instantiated to be a list cell and the code iterates on the tail of the list element. The procedure call to `pushOnResetStack(p3)` is actually a macro which expands into a test which decides whether or not the instantiation of *p3* should be recorded so that it can be undone on backtracking (described in section 4.0, "Backtracking and delaying" on page 66).

This code implements a deterministic Prolog predicate which can either return a result (if the third argument is fully or partially uninstantiated) or test that the third argument is the correct answer, so the query must be slightly different: *p3* is initialized here to be an uninstantiated variable.

```
/* ?- append([1,2], [4,5], L3). */

    struct cell *p1, *p2, *p3;
    p1 = allocListElem(allocInteger(1),
         allocListElem(allocInteger(2), &nilCell);
    p2 = allocListElem(allocInteger(3),
         allocListElem(allocInteger(4), &nilCell);
    p3 = allocVariable();
    append(p1, p2, p3);
```

However, the result is different:



Not only is this second version slower, but it also produces extra "reference" cells.[6] Therefore, the compiler should recognize situations similar to *append* (which occur frequently) and optimize them as described above (this can be considered as a kind of peephole optimization on the abstract machine).

[6] In fairness to the Warren Abstract Machine (WAM), this only happens if my scheme if individual value cells is used; if uninstantiated variables were stored within the heads and tails of list cells (instead of requiring separate cells), then no extra "reference" cells would be required, in this case. However, execution would still be slower because a check still must be made on each iteration for whether the third parameter is an uninstantiated variable, something which is already known (this avoids general unification and testing for whether the instantiation must be recorded on the reset stack).

2.4 Machine code for deterministic append predicate

Appendix A, "Sample machine code" on page 280 has sample machine code which translates the above C code to IBM/370 code. These transformations cannot easily be done by a C compiler, because certain values must be kept globally in registers for ultimate performance.

The best performance can be attained if the abstract machine status registers and the argument registers can be mapped into the target machine's registers. The **xpPAM** registers contain pointers to objects, so they translate in a natural way to IBM/370 registers (see section 3.0, "The basic sequential inference engine" on page 28 for the **xpPAM** status registers). Additional speed is possible, if the target machine allows some kind of object access, dereferencing and tag handling in parallel.

The inner loop of the second (slower) version is 19 or 23 machine instructions, depending on whether heap overflow is detected in-line or by an exception.

The inner loop of the first (faster) version is 8 or 10 machine instructions, depending on whether heap overflow is detected in-line or by an exception. On a "1 MIPS" machine, the above loop will run at over 100 KLIPS (thousands of Logical Inferences Per Second).

The inner loop has the same instruction count, for either allocating from a heap or for allocating from a stack. For reference counting, three extra instructions are needed (load, add one, store).[7]

2.5 Abstract machine code for deterministic append predicate

Skipping ahead a bit, here is the code used by the abstract machine (xpPAM) for a purely deterministic predicate. This abstract machine code can be used to directly generate the C or assembler code given above (a slightly different version of this, with delays, is in section 4.4, "Delays" on page 75):

```
  swXVNL f0, n3, n0 % L1 = X . A
  builtin 3         %    on failure: error
  goto var          %    var
  goto nil          %    []
lst:                %    _._
  eqlst f2, f3, n2  % L3 = X . C
                    % arg0: A (already there)
                    % arg1: B (already there)
                    % arg2: c (already there)
  lCallSelf         % append(A, L2, C)
nil:
  eq f1, f2         % L2 = L3
  return
var: % from here on, set up for backtracking
  pushB v0
  pushB v1
  pushB v2
```

[7] For this particular example, it appears as if reference counting is significantly slower than a marking garbage collector. However, a marking garbage collector must eventually scan the list and it probably will take more than three instructions to mark a cell. See section 7.12, "Reference counts and garbage collection" on page 153.

```
  mkCh  mkch2
  eqlst f0, n3, n0    % L1 = X . A
  goto  lst
mkch2:
  popB  n2
  popB  n1
  popB  n0
  eq f0, []           % L1 = []
  goto  nil
```

The inner loop is three abstract instructions (swXVNL, eqlst, lCallSelf) which can be reduced to two if the lCallSelf is replaced by the swXVNL.

3.0 The basic sequential inference engine

Total grandeur of a total edifice,
Chosen by an inquisitor of structures
For himself. He stops upon this threshold
As if the design of all his words takes form
And frame from thinking and is realized.
— Wallace Stevens, *To an Old Philosopher in Rome.*

Brevis esse laborio,
Obscurus fio.
(I strive to be brief, and I become obscure.)
— Horace, *Ars Poetica*

XpPAM consists of a basic sequential part which can be described almost completely independently of the features which allow backtracking and delaying.

The machine has 32 general registers, three stacks (execution, reset and backtrack) and a heap. The general registers are used for passing arguments. The execution stack is used to save status and registers across calls. The reset and backtrack stacks are used for backtracking and are described in section

4.0, "Backtracking and delaying" on page 66. The heap is used for allocating all objects.

3.1 Objects

All objects are stored the heap: the object table (*oTab*) and the extension area (*xArea*). All objects are first class citizens and are tagged. For simplicity, arrays and I/O objects (stream or direct) are not described here. Their implementation can easily be inferred from section 11.0, "Arrays and I/O done logically and efficiently" on page 222). Any number of types can be added to the machine with no loss of efficiency (they are just extra cases in branch tables). The following are discussed:

type	**description**
uninstantiated	(or not ground) logical variable.
reference	(pointer) to another object which is automatically dereferenced whenever it is accessed.
number	integer or floating point. The implementation has only floating point numbers.[8]
string	commonly called an *atom*.
nil	denoted "`[]`".
list element	containing pointers to two objects, denoted "`Head.Tail`" or "`[Head|Tail]`".

8 This is based on two considerations: it makes implementation easier and some modern machines can do floating point arithmetic nearly as fast as integer arithmetic.

paged out	object with pointer to backing store. Data in backing store never point to data in primary store. See section 3.15, "Virtual memory" on page 61 for details. Paged out objects are automatically paged in when accessed.
code segment	contains abstract machine code (section 3.6, "Code segments" on page 36).
thunk	code pointer with environment [Ingerman 1961] (section 6.4, "Thunks, lazy evaluation and higher order functions" on page 127).
cut point	information for handling a cut.
unallocated	on the free list (only used for heap overflow testing).

XpPAM uses structure copying because its implementation is simpler than structure sharing and it has similar space and time efficiency [Mellish 1982]. There is, however, nothing in xpPAM's design which prevents using structure sharing instead of structure copying.

Each object is identified solely by its address within the object table (*oTab*). The address remains constant throughout the object's lifetime. Each object fits in a fixed size cell (about 12 bytes, depending on the base machine) containing:

- tag (determines the object's type)
- flags (meaning depends on the object's type)
- reference count (or garbage collector marking bits)
- data, consisting of one of:
  - one pointer
  - two pointers

- a numeric value
- pointer into the extension area (*xArea*) plus additional information (such as a hash bucket pointer for strings)
- "age" and delay information for uninstantiated variables

All objects are kept in one heap (*oTab*) with an extension area (*xArea*) for large objects (strings and code). All objects are subject to garbage collection – cells within *oTab* are simply added to the free list and corresponding space within *xArea* is compacted as needed. I implemented a reference counting garbage collector because it allows reclaiming memory as soon as possible. A marking garbage collector could easily have been used instead (see section 7.12, "Reference counts and garbage collection" on page 153).

Numbers, *nil*s and list elements fit entirely within these cells. Strings, code segment cells and thunks contain pointers into a separate area which is divided into segments and is compacted like Smalltalk-80's LOOM (see section 3.15, "Virtual memory" on page 61).

Although new strings can be created by the *concatenate* or *substring* operations, most strings are constants which are known when the code is loaded. The loader ensures that only one copy is kept of each such constant string (fast lookup at load time uses hash values). Two constant strings are equal if and only if they are at the same address. Dynamically created strings require full character by character comparison. Constant strings and dynamically created strings are distinguished by a flag bit.

For simplicity, **xpPAM** treats functors as lists (as in micro-Prolog [Clark and McCabe 1984]) so that, for example, `f(a,b)` $\approx$ `[f,a,b]`. Functors and lists are distinguished by a flag in the head element so that a list element cannot unify with a functor element (however, `F(A)=f(1)` results in `F=f`, `A=1`). See section 7.10, "Functor and list storage" on page 151 for a fuller discussion of alternative representations.

3.2 General registers

The machine has 32 general registers. The number of registers is somewhat arbitrary and does not affect the overall design. [Auslander and Hopkins 1982] note that on a RISC with 16 registers, about half the programs need register spill code; with 32, fewer than 5% need spill code. This statistic, however, may not apply to an inference engine.

Each of the registers contains either an object address or is flagged as being empty. A hardware implementation could keep a shadow cache of the objects referenced by the registers (that is, a register would contain the object's address and the shadow cache associated with the register would contain the 12 or so bytes of the object's value cell).

The general registers are used primarily for passing arguments to predicates. An n-place predicate will receive the addresses of its arguments in registers 0 through $n-1$.

If a predicate has more than 30 parameters, it must be transformed by the compiler to code the last n-30 into a structure. The limit is 30 rather than 32 because some instructions require up to two extra registers (for example, splitting a list element into its head and tail may require two additional registers to hold the head and tail).

```
p(..., P30, P31, P32, ..., Pn)
```

is transformed to

```
p(... , (P30 . P31 . P32 . ... . Pn))
```

3.3 Allocating and freeing object cells

New cells are allocated from a single free list. As all objects are the same size, this list never needs compacting (see section 7.3, "Global and local stacks" on page 142 for comparison with other methods).

"Freeing a register" simply means flagging the register as empty. If reference counting is used, the reference count is decremented and, if it becomes zero, the cell is returned to the free list, possibly causing other cells to be freed. If a marking garbage collector is used, the register must still be flagged as empty so that the marking algorithm knows where to start and so that fail and delay work properly.

3.4 Status registers

In addition to the 32 "general purpose" registers which are mainly used for passing arguments, special registers keep the machine's status. Some of these are needed only for backtracking or delaying and are described more fully in the next chapter.

pc program counter: contains the code segment and offset of the next instruction.

cpc continuation program counter: contains the code segment and offset of the next instruction to be executed after a *return* instruction.

toes top of execution stack.

ptoes protection for (top of) execution stack: *toes* value in the top frame on the backtrack stack.

linkr link register: used by `link`, `call` and `unlink` instructions to save the *toes* (needed for backtracking).

tobs top of backtrack stack.

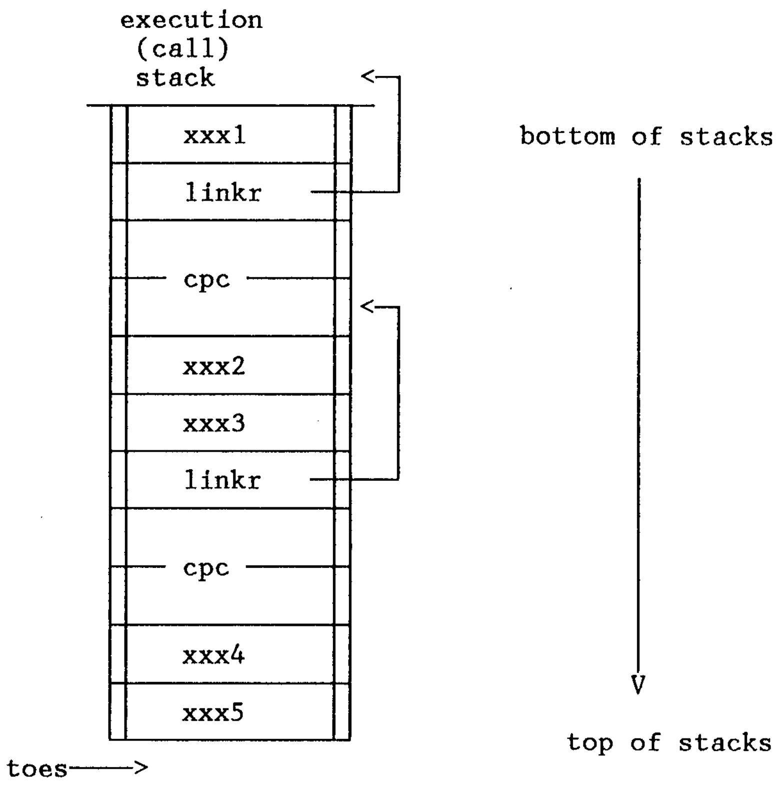
tors top of reset stack.

popbkr pop backtrack register: a flag and pointer into the choice stack which is set by a `popBKeep` instruction (it is set off whenever failure occurs). If this flag is on, a `mkCh` or `mkChAt` instruction will restore the choice stack point to what it was when the failure happened; that is, effectively pushing everything back onto the choice stack.

orwaitr or-wait register: keeps the last "or wait" entry. It is null if no `waitOr` instruction has been done since the last `wait` instruction.

3.5 Execution stack

The execution stack looks like this (growing from the top to the bottom of the page). Section 4.1, "Stacks for backtracking" on page 66 has a diagram showing all the stacks.



The "frames" contain information put there by the `push` (shown as `xxx1` through `xxx5` above) and `call` instructions (shown as `linkr` and `cpc` above). Each entry on the execution stack is one of:

- a value put there by a `push` instruction; or
- a call stack frame:
  - *cpc* in two slots (code segment first, followed by offset),
  - value used by the `unlink` instruction to reset the stack frame after returning from the call; or
- a preemption entry (described in section 4.4, "Delays" on page 75).

The back pointers are not needed if strictly deterministic computation is done. They are needed to allow backtracking to "protect" parts of the stack.

3.6 Code segments

XpPAM instructions are kept within *code segments.* Each code segment corresponds to a single compiled predicate. It contains:

- object address of the predicate's name.
- number of parameters.
- size of the constants' vector.
- size of the code vector.
- debugging information.
- pointer to source (for debugging).
- constants and code. (The constants' vector contains object pointers for all the constants used within the code. The code follows immediately after.)

The constants' vector contains pointers to objects. The loader always handles pointers to code segments by inserting an indirection via a reference cell; this allows replacing a code segment dynamically.

3.7 Machine instruction format

A xpPAM machine instruction fits within 4 bytes. The three operands take 18 bits (6 bits each) and the opcode plus flags can be encoded within the remaining 14 bits (the exact encoding is not important and will not be discussed here). In the emulator, a looser packing is used for efficient extraction: the opcode is 1 byte, the flags are in 2 bytes and the three operands take 3 bytes (1 byte each). Some instructions could be packed tighter but such packing is not worth the extra time needed to extract fields.

The two formats are:

- three operands: *Op1*, *op2* and *op3* are each a register number or a constant number; the meaning is determined by the flag values. For some instructions, only one or two of the operands are used.

opcode	flags	op1	op2	op3

- one operand and offset: *Op1* is a register number or a constant number; the meaning is determined by the flag values. For some instructions, *op1* is not used.

opcode	flags	op1	offset

A "register number" is a 1-byte quantity with a value from 0 through 31. A "constant number" is an index into the constants' vector for the code segment with a value from 0 through 63.

In instructions, each register is annotated:

v contains a value.

n empty, possibly requiring the allocation of a new object.

f contains a value which is emptied after use.

x is empty and the value is unneeded (*n*+*f*). This could be used, for example, when a head or tail value is not needed (the anonymous variable "_" in "`_.Tail`").

c constant (index into the code segment's constants' vector).

s (for the `eqlst` or `swXVNL` instructions' first operands only) means structure head rather than list element.

V, *n* and *c* are mutually exclusive. The flags can be encoded by combining them with the opcodes – most instructions have only one or two possible flag combinations, so this would be the fastest possibility, at the expense of having many opcodes (the software emulator uses a slower method, by separately decoding the flags).

F annotations mark where reference counts are decremented (if a marking garbage collector is not used). Whether or not reference counting is used, registers must still be flagged as empty to minimize the amount of information stored when choice points or "thunks" are created.

Each code segment has a vector of up to 64 constant objects' addresses. Most instructions allow *c* annotations to indicate that the operand is the index of an entry in the constants' vector.

3.8 Net effect of calls

When an n-ary predicate is called, the caller must save its registers on the execution stack and put the arguments in registers 0 through $n-1$. The called predicate must ensure that on return all registers are empty – the caller then pops the stack to restore the saved registers. That is, on entry to an n-ary predicate only the first n registers are non-empty. On return, all the registers must be empty – the called predicate must explicitly free them.

3.9 Tail recursion optimization (TRO)

The *pc* and *cpc* are used to allow tail recursion optimization. The *cpc* contains a return address. A call must

- Push the active registers onto the stack.
- Push *cpc* onto the stack.
- Copy *pc* into *cpc*.
- Set *pc* to be the first instruction of the called predicate.

A return must do the reverse:

- Copy *cpc* into *pc*.
- Pop the stack into *cpc*.
- Pop the saved registers from the stack.

In a *call* immediately followed by a *return* there is no need to do this push/pop sequence. A *last call* instruction acts just like a *go to*. The *cpc* is left alone so that the called predicate will return to where the calling predicate would have returned.

Tail recursion optimization works only because parameters are passed in registers and not on the execution stack. If the last call is to the same predicate, the TRO works like iteration, with all the values' addresses in registers.

3.10 Unification

Unification is one of the fundamental concepts in logic programming. The abstract machine has several instructions which implement this. The essence of unification is:

- If either operand is a reference cell, it is repeatedly dereferenced.
- If both operands are atomic (numbers, strings or *nil*), they are compared for equality (which may fail). If the strings were known at load time (not created dynamically), they are compared by comparing their addresses (the loader ensures uniqueness).
- If both operands are uninstantiated variables, the newest one is changed into a reference cell which points to the other operand (and possibly recorded on the reset stack). The "age" information is used for this (see section 4.1, "Stacks for backtracking" on page 66 for more details).
- If one operand is an uninstantiated variable, it is changed into a reference cell which points to the other operand (this instantiation is recorded on the reset stack if the variable is older than the latest choice point).

- If both operands are list elements or functor elements, unification is done recursively on the heads and tails of the operands.
- Otherwise, unification fails.

Note that the above definition guarantees that a reference cell is always newer than the cell it points at, so there can never be any dangling pointers.

Unification does not do the occurs check (see section 10.6, "The occurs check" on page 217). There is no separate push-down stack for unification and deallocation because the Deutsch-Schorr-Waite algorithm is used [Knuth 1973] to traverse lists by reversing pointers using special tags which are not normally visible. These special tags are also used to terminate an infinite unification.

3.11 Instructions

Some instructions may fail or delay. These concepts are described in the next chapter.

The assembler notation is a symbolic opcode followed by up to three operands. Each operand is a single letter (its annotation: *v*, *n*, *f*, *x*, *c*, *s*) followed by the register number or constant number.[9]

[9] Just to confuse you, the assembler has the single letter after the register number so that "eq n1,f2" is input "eq(1.n,2.f)"

The instructions are given with pseudo-code for interpreting them. There are five steps in interpreting:

- Decode the opcode.
- Pre-process the operands into internal registers *r1*, *r2* and *r3*.
- Increment the program counter (pc = pc + 1).
- Execute the instruction.
- Post-process the operands in the actual registers.

As mentioned above, every operand is one of:

- A register number
- A constant number
- An offset value

For the first two cases, a value is developed in one of the internal registers *r1*, *r2* or *r3*, corresponding to operands 1, 2 or 3, according to the annotation:

Annotation	**Action**
v, f, s	Internal register receives the appropriate register's contents, then dereferences it if necessary.
n, x	Internal register receives the address of a new uninstantiated variable cell.
c	Internal register receives the address of the appropriate constant (by indexing into the code segment's constants' vector).

For most instructions, the internal register values are repeatedly dereferenced until a non-reference object is reached (the unification algorithm guarantees that there is never an infinite cycle of references). Exceptions include the *push* instructions.

After the instruction, the operands are processed according their annotation:

Annotation	**Action**
v, n, s	Do nothing (the results are already in the general register).
f, x	Decrement the object's reference count and mark the general register as being empty (this is not done for some cases of some instructions, such as swXVNL).
c	Do nothing (it's not a register).

The internal registers are necessary to allow certain "overlapping" register combinations. For example, it is often convenient to have a list element (object pointer) in register 0 and then replace register 0 by the list element's head (the tail goes to another register). This can be accomplished by a single instruction (e.g., `eqlst f0,n0,n5`).

The internal registers can not be blindly copied back to the operand registers because the internal registers may have been dereferenced – a future unification may need to point to the un-dereferenced object. Therefore, the instructions must operate directly on the general registers, using the values in the internal registers for input. Instructions must also increment reference counts as necessary.

The annotations *c*, *v* and *n* are mutually exclusive. The pseudo-code does not show any test for this error – the assembler assures against invalid annotations.

The pseudo-code is not necessarily optimal. For example, the pseudo code for `eq n1,v2` has the sequence:

- Allocate a new cell for an uninstantiated variable and put its address in register 1.
- Unify the objects pointed by registers 1 and 2.
- Decrement the reference count of the object pointed by register 2 and mark register 2 as empty.

But the implementation simply copies the contents of register 2 into register 1.

Failure handling is described in section 4.2, "Backtracking instructions" on page 70. In the pseudo-code, failure is handled by calling `doFail()`.

Pseudo-code does not show pushing instantiations onto the reset stack (discussed in the next chapter) nor does it show adjusting reference counts.

3.11.1 Unification and testing instructions

eq *op1*, *op2*

Unifies the two operands (which may fail). The operands can be constants or registers. `Eq` can be used to unify values or to move or copy registers. Examples:

`eq n1,f2` moves register 2 into register 1,

`eq n3,v4` puts a copy of register 4 into register 3.

`eq n5,c3` loads register 5 with the address of the 3rd constant in the code segment.

`eq v1,f2` unifies register the object pointed by register 1 with the object pointed by register 2, then frees register 2.

Pseudo-code:

```
if (! unify(r1, r2))
    doFail()
```

eqskip *op1*, *op2*

Like `eq` except that it skips the following instruction if the unification (usually an equality test) succeeds (`eq` fails to the most recent choice point — described in the next chapter). If the equality test fails, any *f* annotations are ignored. Failure does not undo instantiations caused by unification so there will usually be some tests before `eqskip` to ensure that the operands are not variables. This instruction always appears to succeed (never "fails").

Pseudo-code:

```
if (unify(r1, r2))
    pc = pc + 1 /* pc already points at next instruction */
```

testskip *op1*, *op2*

Tests if the two operands are equal. `Testskip` is the same as `eqskip` except that it will not instantiate anything; if instantiation would occur, `testskip` does a *delay* for the uninstantiated variable (section 4.4, "Delays" on page 75).

eqlst *op1*, *opHd*, *opTl*

Unifies the first operands to a list element composed of the second and third operands. May fail.

Examples:

`Eqlst f1,n2,n3` tests for register 1 containing (the address of) a list element (or, if it is a logical variable, instantiates it to a list element) with the head being put in register 2 and the tail in register 3. Register 1 is then emptied.

`Eqlst f0,n0,n1` replaces register 0 by the list element's head (the tail goes into register 1).

Pseudo-code:

```
if (r1 points to a list element) {
    if (! unify(r1->head, r2)) {
        doFail()
    } else if (! unify(r1->tail, r3)) {
        doFail()
    }
} else if (r1 points to an uninstantiated variable) {
    rx = new list element cell with head = r2, tail = r3
    r1->tag = reference     /* r1 points at     */
    r1->ptr = rx            /* new list element */
} else {
    doFail()
}
```

3.11.2 Control instructions

swXVNL *op1, opHd, opTl*

Jumps to one of the following four instructions that immediately follow it, according to the type of the first operand:

Atom (none of the following). The second and third operands are completely ignored (including *f* annotations).

Variable. The first operand is not freed even if it has *f* annotation. The second and third operands are completely ignored (including *f* annotations).

Nil. The second and third operands are completely ignored (including *f* annotations).

List element. The second and third operands are unified with the head and tail of the list element. If during unification, some uninstantiated variables become instantiated but the entire unification fails, the variables are not reset to uninstantiated.

Normally, this instruction is used to split a list element, the second and third operands having *n* annotation. The four instructions following the `swXVNL` instruction are usually `goto`s or `builtin 3` ("error"), except for the last one. Other switch instructions are possible for multi-way branching on strings or numbers; they can be emulated by sequences of `eqskip`s.

The first operand must have the annotation *v* or *f* (*n* is not allowed). *N* annotations for the second and third operands are treated like *v* annotations if the first operand is a variable or atom.

The first operand must have the annotation *s* if it is to handle a structure element.

Pseudo-code:

```
if (r1 points to a uninstantiated variable) {
    pc = pc + 2; /* case var: don't free */
} else if (r1 points to a list element) {
    if (unify(r1->head, r2) && unify(r1->tail, r3))
      then pc = pc + 1; /* "fail" without undoing any instantiations  */
      else { pc = pc + 4;
             copy r2, r3 into registers;
             free registers as needed
      }
} else if (r1 points to nil) {
    pc = pc + 3; /* case []: don't free r2, r3 */
} else {
    pc = pc + 1; /* case default: don't free */
}
```

caseGoto *op1, number*

This is a generalization of swXVNL. The *number* is the number of cases which follow (it is encoded like an offset). CaseGoto dereferences the operand. In the next instruction space are two offsets:

1. where to go if the operand is an uninstantiated variable.
2. where to go if none of the other cases match (default case).

Following are the cases. Each consists of an object pointer and an offset. If the constant in a *case* matches the operand given by the caseGoto, execution continues at the indicated offset within the current code segment. If more than one case matches, it is unspecified which one is taken (this allows hashing or other non-sequential selection).

goto *offset*

Unconditionally goes to the offset within the code segment if the first operand is an uninstantiated variable.

Pseudo-code:

```
pc = offset
```

varGoto *op1*, *offset*

Goes to the offset within the current code segment if the first operand is an uninstantiated variable.

Pseudo-code:

```
if (r1 points to an uninstantiated variable)
    pc = offset
```

nonvarGoto *op1*, *offset*

Goes to the offset within the current code segment if the first operand is not an uninstantiated variable.

Pseudo-code:

```
if (r1 does not point to an uninstantiated variable)
    pc = offset
```

3.11.3 Call / return instructions

push *op1*

Pushes the operand onto the execution stack. This instruction is normally after a `link` and before a `call`. The operand is not dereferenced before pushing.

Pseudo-code:

```
*toes++ = r1 /* copy r1 to top of stack; increment stack pointer */
```

pop *op1*

Pops the operand from the execution stack. This instruction is normally after a `call` and before an `unlink`. The operand must be a register with the *n* annotation.

Pseudo-code:

```
r1 = *--toes /* copy top of stack to r1; decrement stack pointer */
```

gete *op1,offset*

Gets the operand from a specified offset within the execution stack. This instruction is normally used for register spill code. The operand must be a register with the *n* annotation.

Pseudo-code:

```
r1 = toes[0-offset]
```

link

Saves *toes* in *linkr*. If the *ptoes* (protection execution stack) value is above the *toes* value, *toes* is set to the protection value.

Pseudo-code:

```
linkr = toes
if (toes < ptoes)
    toes = ptoes
```

unlink

Restores *toes* from *linkr*.

Pseudo-code:

```
toes = linkr;
```

return

Restores the machine registers to their state when the `call` was executed (see description under `call`).

Pseudo-code:

```
x = *toes--; /* pop top element of execution stack into "x" */
if (x points to code) {
    pc = cpc;
    pc = *toes--; /* (pop 2 elements) */
    linkr = *toes--; /* pop link register */
} else {
    do special preempt or delay return (described in next chapter)
}
```

call *op1*

calls (tries) a predicate. There are two varieties:

- `call c`*C*: call a known code segment whose address is given by the constant *C*. The argument registers are assumed to be already loaded. Note that the constant is usually a reference to a code segment (this is what the assembler generates).
- `call f`*R*: dynamic call – *R* points to either
  - The argument list of which the name is the first element. The name is looked up and the registers are loaded with the elements of the list.
  - A thunk. The thunk specifies the first *m* of an *n*-ary predicate. The first $(n - m)$ registers are shifted right *m* places and the first *m* registers are loaded from the thunk. The register should have *f* annotation to ensure that it is freed before the registers are loaded with the arguments.

Cpc and *linkr* are pushed onto the execution stack, *pc* is copied into *cpc* and *pc* is set to the first instruction in the code segment. `Return` does the inverse by copying *cpc* into *pc* and popping the execution stack into *cpc* and *linkr* (all the registers must be empty when a *return* is done). Using *cpc* this way allows the last call (`lCall`) instruction for tail recursion optimization.

Pseudo-code:

```
*++toes = linkr; /* push linkr */
*++toes = cpc; /* push cpc (two elements) */
cpc = pc; /* cpc = next instruction */
if (r1 does not point to code segment) {
    if (r1 points to an atom) { name = r1; n = 0; }
    else if (r1 points to a list element or structure element) {
        name = r1->head;
        n = number of elements in r1's list;
    } else error;
    search for code for name/n predicate
       and fill the registers 0 to n-1 with the arguments
    pc = first instruction in thunk's code segment;
}
pc = first instruction in code segment;
```

When a predicate is searched for, and it has not been created, a new predicate is created which calls "<undef>." The <undef> predicate can be tailored to the user's needs, for example, to just fail (as in conventional Prolog), to output an error message ("undefined predicate") or to prompt for a definition (a "query the user" facility).

The operand for a dynamic call may be a functor or a list. XpProlog allows call([Name,Arg]) instead of Name(Arg) (avoiding the need for *univ* ("=..") for constructing a functor from a list.

callSelf

Like call except that there is no need to switch code segments (a slight optimization).

lCall *op1*

Last (tail recursive) call. This is the same as call except that it pushes nothing onto the stack.

lCallSelf

Last (tail recursive) call to self. This is the same as `goto 0`. It is included to help in debugging.

Pseudo-code:

```
pc = 0
```

3.11.4 Allocation / deallocation instructions

free *op1*

Frees the operand. The operand must be a register with *f* annotation. The reference count of the object pointed at by the operand is decremented and the register is marked as free. This instruction is seldom needed because *f* annotation in other instructions can free registers.

new *op1*

Allocates an uninstantiated variable in the operand (the register must be empty). The operand must be a register with *f* annotation. This instruction is used mainly for setting up arguments which return values from predicates; however, the `push` is usually used instead with `n` annotation.

3.11.5 Miscellaneous instructions

builtin *number*

The *number* is coded as an offset operand which identifies the built-in. This is used to extend the machine's instruction set with "foreign" instructions such as arithmetic, string manipulation, i/o, etc. The various built-in instructions are somewhat idiosyncratic in how they use registers. They are described in an appendix. A built-in may fail or delay.

stop

Stops the top level interpreter.

label

a pseudo-instruction for the assembler.

comment

a pseudo-instruction for the assembler – ignored.

3.12 Code example

The (not very useful) example:

```
p([], []).
p(a.Rst, x.OutRst) :- p(Rst, OutRst), q(a, Rst).
p(b.Rst, y.OutRst) :- p(Rst, OutRst).
```

is compiled to:

```
  swXVNL    f0,n2,n0  % switch on parm0: Hd=>reg2, Rst=>reg0
  fail                % invalid parm
  goto      var       % variable
  goto      nil       % parm0=[]
lst:                  % parm0=Hd.Rst
                      %   Hd  is now in reg 2 (from swXVNL)
                      %   Rst is now in reg 0      --"--
  eqskip    f2,'a'    % test Hd = 'a'
  goto      else      %   fail test .. go to "else"
  eqlst     f1,'x',n1 % 'x'.OutRst
  link
  push      v0        % save Rst
  call      p/2       % p(Rst, OutRst)
  pop       n1        % restore Rst into required reg
  unlink
  eq        n0,'a'    % first arg for q/2
                      % second arg already in reg 1
  lCall     q/2       % q(a, OutRst)
else:
  eqskip    f2,'b'    % test Hd = 'b'
  fail                % else: invalid parm
  eqlst     f1,'y',n1 % 'y'.OutRst
  lCallSelf           % p(Rst, OutRst)
nil:
  eq        f1,'[]'   % result := []
  return
var:
  ... code left out for now
```

Note: The notation `eqskip f2,'a'` is not supported by the assembler.[10] Instead, the program would have to explicitly list out the constants: a, x, b, y, []

[10] To allow the assembler to easily handle labels, a slightly different notation is used: the register number is given first, followed by the annotation. For example, `eq n1,c2` would be coded `eq(1.n,2.c)`.

and `call(p,2)`.[11] Using this, `eqskip f2,'a'` would be presented to the assembler as `eqskip(2.f,0.c)` (the constant "a" is the 0th constant) and "`call p/2`" would be presented as `call(5.c)`.

The code for handling a variable for the first parameter has been left out. The above code implements a deterministic predicate – if backtracking code were added, it would not affect the efficiency of the deterministic code. Backtracking code is described in section 4.0, "Backtracking and delaying" on page 66 and section 5.1, "Unoptimized compiling of a predicate" on page 88.

If this were marked as a "never fail" predicate, failure should give an error indication. To do this, the `fail` instructions could be replaced by `builtin 3` ("error") instructions.

The `lCallSelf` instruction has the same meaning as `goto 0` (the different opcode helps in debugging). This is a tail recursive call – recursion has been turned into iteration.

[11] `call(p,2)` is used by the assembler and loader to look up the 2-arity "p" predicate. A reference to the predicate's code segment is created. If the predicate p/2 has not yet been defined, it is created, calling the built-in *undef* which can use information from the registers to print out an error message or prompt for more input, before failing.

3.13 User defined unification

The current implementation implements simple syntactic unification. It does not require a local stack because pointer reversing (the Deutsch-Schorr-Waite algorithm) can encode the stack in the pointers (this algorithm has another advantage: because it changes tags on its way "down," infinite loops can be detected). Another possible implementation is to have the unification instructions handle only atomic values and to invoke a separate `eqXX` predicate for compound values (that is, list elements). This is defined:

```
eqXX(X, Y) :- atomic(X), X=Y.
eqXX(H1.T1, H2.T2) :- eqXX(H1,H2), eqXX(T1,T2).
```

which is compiled into

```
   swXVNL f0,n0,n2
   goto   atomic    %  atomic
   goto   atomic    %  var
   goto   atomic    %  []
lst:                %  _._
   eqlst  f1,n1,n3
   link
   push   f2        % save T1
   push   f3        % save T2
                    % 1st arg already in r0
                    % 2nd arg already in r1
   callSelf
   pop    n1        % T2
   pop    n0        % T1
   unlink
   lCallSelf
atomic:
   eq     f0,f1     % guaranteed atomic/var 1st operand
   return
```

A similar predicate would be needed for the `testskip` instruction, being careful to not cause any unifications. This simplifies the abstract machine because unification is much simpler, possibly speeding up execution. Most unifications are for very simple cases, so this `eqXX` is seldom called. The intriguing thing about this design is that allows easily changing the unification to implement any kind of equality. In this way, functional programming could be easily added to a logic programming language. Also, new user data types can be added and integrated into the standard unification mechanism.

3.14 Other object data types

[Mukai and Yasukawa 1985] propose *complex indeterminates.* These have many similarities with arrays, records and "frames" except that they are of indeterminate size. Here is a sample complex indeterminate:

```
<f1/X, f2/abc, f3/[a,b]>
```

This has three *fields*: `f1`, `f2` and `f3` with associated data (`X`, `abc` and `[a,b]`, respectively). Field names must be strings; they may be given in any order. Here are some sample unifications:

```
<f1/X, f2/abc> = <f2/abc, f1/Y>   ==> X = Y
<f1/X, f2/abc> = <f2/X>           ==> X = abc
A=<f1/X>, B=<f2/abc>, A=B         ==> A = B = <f1/X, f2/abc>
```

A new field can be "added" to a complex indeterminate, as shown by the last example. Complex indeterminates are very useful for keeping associative information, such as symbol tables, syntax parts of a sentence or frames (for AI problem solving).

Complex indeterminates have not been implemented in xpPAM. I will probably be implement them as binary trees. Of course, they do not need to be built into xpProlog; the user could explicitly unify them. But it is nicer to have them built-in. If they are not built-in, user-defined unification can achieve the same affect.

New built-in object data types such as complex indeterminates can easily be added to xpPAM. Three changes must be made:

- The storage allocator/deallocator must be modified. This is very simple, consisting mainly of code to compute the size needed in the extension area *xArea.*
- The unification algorithm must be modified. This consists mainly of adding new cases. Thus, new object data types do not slow anything down.
- New unification instructions (for example, similar to `eqlst` or new switch instructions) to handle the extra data types.

To alleviate the work required to add new object data types, xpPAM could be modified to have more general forms for some instructions. For example, instead of swXVNL, there could be a more general switch statement which is simply followed by a list of tags and branch offsets.

3.15 Virtual memory

The virtual memory system is similar to Smalltalk's LOOM (Large Object-Oriented Memory) [Goldberg 1983]. As in most Smalltalk implementations, I have used reference counting (see section 7.12, "Reference counts and garbage collection" on page 153).

The heap uses an object table called *oTab* which is a vector of fixed size entries, each containing the type flag, reference count and a variant part depending on the type. Numbers, *nil*, list elements, uninstantiated variables and references can all be contained entirely within a single *oTab* entry's variant part. For strings and code, the variant part is a pointer to a variable size second part which is allocated in an extension object area *xArea*.

Functors could be stored as separate structures in the extension area (*xArea*). See section 7.10, "Functor and list storage" on page 151.

Every object is referred to only by its index in *oTab* which is called an "address." For efficiency, the *xArea* part of an object may be locked for a short time to prevent its being moved by the memory manager (this is normally only for code segments) but normally the memory manager may reorganize the items in *xArea* at any time so long as it updates the pointers from *oTab* into *xArea*.

Because register contain only object addresses (pointers into *oTab*), the extension data for an object (in *xArea*) can be moved at any time, provided the pointer from the object is updated.

The *xArea* area is divided into fixed size segments. Whenever a new object cannot be allocated in the current segment, the next segment is compacted and the object is allocated in that segment. Because most objects have short lives, compaction usually moves only a few items – objects with long lives sink to the bottom of the segments and tend not to be moved. There is no noticeable delay when a segment is compacted because segments are relatively small.

To aid in compacting *xArea*, each object in *xArea* has a pointer back to the object in *oTab* which needs it. Compaction is done by stepping through the objects in a *xArea* segment, skipping over any objects with null back pointers (freeing an object sets the back pointer to null, in addition to returning its *oTab* cell to the free list). If there is a locked object in the segment, compaction starts after it (the implementation allows only one locked object at a time).

The unallocated objects are kept in a free list within *oTab* with new objects being allocated from the front of this list. All value cells except strings and code can be allocated entirely within single *oTab* entries. Code is fairly static and strings are only allocated and deallocated by explicit string operations like "concatenate" and "substring," so allocating and deallocating in xpPAM's heap is as efficient as pushing and popping a stack because the *xArea* area will seldom need compacting (see section 7.3, "Global and local stacks" on page 142).

3.16 Object paged virtual memory

Full virtual memory design can be implemented similar to that in [Goldberg 1983]. Objects rather than fixed size pages are moved on demand between main memory and secondary memory. Objects in secondary memory may have a different representation than objects in primary memory – for example, pointers could be larger in secondary memory, or functors could be kept as records instead of as linked elements (similar to "cdr-coding").

Objects in secondary memory never point to objects in primary memory. Objects in primary memory may point to objects in secondary memory. If an object in primary memory has an image in secondary memory, the primary image has a pointer to secondary memory – when an object is paged out, it goes to its original place in secondary memory.

When an object containing an address (for example, a list element or a reference object) is moved into primary memory, the addresses within it are set to point at special "in secondary memory" objects which contain the secondary memory addresses. Whenever such an object is used, **xpPAM** brings the object in from secondary memory. It is possible that an object in secondary memory is already in primary memory. To aid in finding this, *oTab* entries have an additional field which is the secondary memory address (null if the object is only in primary memory). The *oTab* entry for an "in secondary memory" object is picked by hashing the secondary memory address (this requires that the free list be forward and backward chained so that objects can be allocated from the middle of the list). In this way, an object can be brought in from secondary storage and efficiently transformed into its internal form.

A special "in secondary memory" object is used to reduce the number of entries in *oTab*. There is only one instance of this object. Whenever it is referenced, **xpPAM** cannot directly tell which secondary memory object is needed – it must fetch a fresh copy of the secondary memory object which points at this object, and use the secondary memory pointers contained there. This process can be optimized – the details are explained in [Krasner 1983].

Reference counts are slightly different for paged memory. Each object in secondary memory has its own reference count. The reference count in primary memory is the number of primary memory references to the object. In addition, a delta reference count is needed to record the conversions between secondary memory pointers to primary pointers. The actual reference count for an object is the sum of these three numbers (again, for details, see [Krasner 1983]).

To allow paging out unused objects, each *oTab* entry has a "recently used" bit. Whenever objects must be paged out, those with this bit set off are copied out and all the bits are set off. This is a primitive version of "least recently used." Clearly, hardware assist would improve the efficiency of this operation.

I have gone to the trouble of designing a separate virtual memory system because small machines do not provide virtual memory and I want **xpPAM** to run well on them, too. [Krasner 1983] points out that implementation which rely on a large heap in a standard paged virtual memory tend to degenerate to poor locality of reference so that eventually almost every memory reference produces a page fault and slows execution. However, there is a cost to this virtual memory scheme: *oTab* entries must be larger to contain the secondary memory information and there will be about a 10% overall execution overhead.

Even when gigabyte real memories become reality, some form of virtual paged memory is still likely to be needed. The above techniques are merely suggestions about how object paged virtual memory can be added to **xpPAM** without affecting its design very much. Anyone who wishes to implement large address spaces should review the literature – research is progressing rapidly.

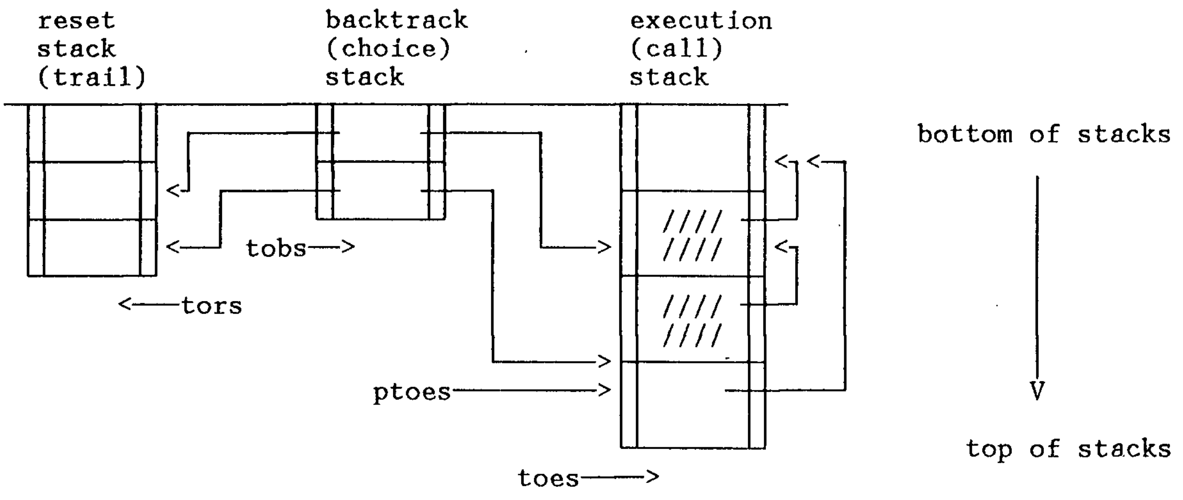
4.0 Backtracking and delaying

O! call back yesterday, bid time return.
— William Shakespeare, *Richard II, act 3*

The deterministic inference machine is easily extended to allow backtracking by adding the backtrack (choice point) stack and reset stack ("trail").

4.1 Stacks for backtracking

The three stacks look like this (growing from the top to the bottom of the page):



Execution frames marked "////" are frames which are "protected" by the backtrack stack (using *ptoes*). In a stack machine for a conventional language like Pascal, these would have been reclaimed on procedure returns; with **xpPAM**, these are reclaimed only if the *ptoes* doesn't protect them. (described later in this section). The back pointers on the execution stack are used to skip over these protected pieces on return from predicates. Such back pointers are common on conventional machines although they can be avoided – for backtracking, they are essential.

The contents of the reset stack are put there by

- unification, when a variable becomes instantiated:
  - object's age (pushed last)
  - object address
- delaying and resuming predicates:
  - special value to indicate a delay (pushed last)
  - object address

When an uninstantiated variable cell is created (by a *new* instruction or by *n* annotation), its "age" is recorded. This age decides whether or not information will be pushed onto the reset stack when the cell becomes instantiated. The "age" is the depth of the backtrack stack when the cell was created. When unification instantiates a value cell which is older than the top choice frame, the cell's address is put on the reset stack. If the cell is newer, backtracking will free it anyway, so there is no need to record its instantiation. Deterministic predicates do not create reset stack entries because such predicates do not create choice points. Any new variables created within a deterministic predicate are newer than the last choice point.

For non-deterministic predicate, choice points must be created on the backtrack stack using the `mkCh` ("make choice point") instruction. Each choice point frame contains sufficient information to reset the machine to the state it was in when the `mkCh` was executed:[12]

- where to go on failure (code segment pointer and offset) (pushed last),
- *cpc*,
- depth of reset stack,
- execution stack protection point,
- execution stack depth,
- non-empty registers (put there by `pushB` instructions).
- "age" of the previous backtrack entry (so that cut instructions leave the ages of uninstantiated variables correct).

If a choice point or continuation point is pushed onto a stack, the code pointer is always pushed on last. This is so that debugging and tracing can determine what is going on (there is no guarantee that, for example, code offsets and object pointers have disjoint ranges). Choice point frames are of variable size, depending on the number of `pushB` instructions used to save registers.

---

12 Actually, the "age" of the previous choice point must also be recorded if cuts are allowed, so that instantiations are properly recorded on the reset stack. Cuts can remove entries on the backtrack stack, so the "virtual choice depth" must be saved in backtrack entries.

When failure occurs, by a unification failing or by an explicit fail instruction, all registers are emptied and the top choice point frame is used to fill the registers. The reset and execution stacks are popped to what they were when the choice point was created. As the reset stack is popped, its entries are used to reset objects to uninstantiated (the age is recovered from the reset stack).[13] Execution then resumes at the failure instruction address.

When backtracking occurs, the execution stack must be restored to what it was when the choice entry was made. This means that a *return* instruction may not pop the execution stack if a choice entry needs it – the choice entry "protects" the entry in the execution stack [Gabriel et al 1985]. The top of execution stack value in the top choice point frame is used to determine whether the execution stack can be popped. Each frame in the execution stack has a back pointer to the previous frame, skipping frames which are protected by the choice stack. If only deterministic predicates are executed, nothing is put onto the choice stack and the execution stack grows and shrinks just like the execution stack in a conventional machine (Algol, Pascal, etc.).

The backtrack stack could be embedded in either the execution or the reset stack. The WAM combines the backtrack stack with the execution stack by threading its entries among the execution entries – I would prefer to combine it with the reset stack because both stacks handle information which is only used

13 Destructive assignments can be handled by keeping a pointer to the previous value instead of just keeping the object's age. This can be useful for implementing compound objects such as arrays (see section 11.0, "Arrays and I/O done logically and efficiently" on page 222) in which changes are recorded on the reset stack.

on backtracking. For clarity of explanation and simplicity of implementation, the stack are kept separate.

Simple instructions improve performance, even for interpreters. Therefore, a "call" is several instructions:

```
  link       % toes skips over protected frames
  push  ...  % one push for each saved value
  call
  pop   ...  % one pop  for each saved value
  unlink     % reset toes below protected frames
```

Similarly, "make choice point" is coded:

```
  pushB ...  % one pushB for each non-empty register
  mkCh label % push the failure address, toes, tors
     ...
label:
  popB ...   % one popB for each saved register
```

For implementing *if-then-else*, the `mkChAt` instruction saves the *tobs* value in the indicated register. A later `cutAt` ("hard cut") pops the choice stack (and reset stack) back to the designated choice point; a `rmChAt` ("soft cut") changes the failure address to point to a fail instruction (if the choice point is at the top of the backtrack stack, the choice point is removed instead).

4.2 Backtracking instructions

Some men a forward motion love,
But I by backward steps would move.
— Henry Vaughan, *The Retreat, l. 29*

pushB *op1*

Pushes the operand onto the backtrack stack. The operand usually has *v* annotation (in which case the reference count is incremented) or *f* annotation.

Pseudo-code:

```
*tobs++ = r1 /* copy r1 to top of stack; increment stack pointer */
```

popB *op1*

Pops the operand from the backtrack stack. The operand must be a register with the *n* annotation.

Pseudo-code:

```
r1 = *--tobs /* copy top of stack to r1; decrement stack pointer */
```

popBKeep *op1*

Pops the operand from the backtrack stack and sets a flag so that a subsequent `mkCh` or `mkChAt` instruction will reset *tobs* to the value it had when the last failure occurred (failure causes this value to be saved). The operand must be a register with the *n* annotation.

Pseudo-code:

```
r1 = *--tobs; /* pop backtrack stack */
popbkr->flag = true; /* set popBKeep happened flag on */
```

mkCh *offset*

Makes a choice point so that backtracking will continue at the instruction specified by the offset. Pushes *toes*, *ptoes*, *tors*, *cont* and *offset* onto the backtrack stack. If *ptoes* is below *toes*, it gets set to *toes*.

Pseudo-code:

```
if (popbkr->flag) { /* was there a popBKeep instr? */
   tobs = popbkr->ptr; /* restore value put there by fail */
   popbkr->flag = false; /* turn off flag */
}
*tobs++ = toes;
*tobs++ = ptoes;
*tobs++ = tors;
*tobs++ = cpc; /* two slots */
*tobs++ = pc;  /* two slots */
if (ptoes < toes)
   toes = ptoes;
```

mkChAt *op1*, *offset*

Makes a choice point and saves information for a subsequent `rmChAt` or `cutAt`. Same as `mkCh` but additionally creates a cut-point entry containing the *tobs* value which is put into *op1* (which must be a register with *n* annotation).

rmChAt *op1*, *offset*

Removes a specified choice point. *Op1* must be a register containing a cut-point entry (from `mkChAt`); the stack frame on the backtrack stack has its continuation point changed to that given by *offset* – usually, this is a series of `popB` instructions followed by a `fail` instruction. This instruction is used for "soft cuts."

cutAt *op1*

Cuts the choice stack back to a specified choice point. *Op1* must be a register containing a cut-point entry; the backtrack stack is popped to this point (as if a series `fails` were done). This instruction is used for "hard cuts."

chopBack

Removes all unnecessary backtrack points from the top of the backtrack stack. Each backtrack "frame" whose reset stack pointer is above the top of the reset stack can be removed because the choice point did not instantiate anything, so no new information will be produced on backtracking.

fail

Causes an unconditional failure. All registers are freed and the backtrack point is popped – the reset stack and execution stacks are popped to the points recorded on the backtrack stack (entries on the reset stack are used to un-instantiate variables or to put delay entries back onto the delay queue), the registers are loaded from the information in the backtrack stack, and execution proceeds at the place stored in the backtrack frame.

Pseudo-code:

```
free all registers;
pc    = *--tobs; /* two slots */
cpc   = *--tobs; /* two slots */
tors  = *--tobs;
ptoes = *--tobs;
toes  = *--tobs;
uninstantiate or return to delay queue from reset stack back to "tors"
   (2 slots each: one for object, one for object's age);
free entries on execution stack back to "toes";
popbkr->flag = false;
popbkr->ptr  = tobs; /* remember backtrack stack position */
```

4.3 Cost of backtracking

Deterministic predicates run slightly slower on the full backtracking machine than on a purely deterministic machine. There are four overheads:

- making choice points rather than just branching to a failure address for *if-then-else.*
- `link` and `unlink` instructions, which are not needed for deterministic execution (nor are the execution frame back pointers).
- testing whether an instantiation should push an entry onto the reset stack (for deterministic execution, nothing will ever be pushed); similarly for recording delay information on the reset stack.

The first two items can be avoided by a smart compiler, using the switch instructions or by using knowledge of such predicates as "$=$," "$\neq$," "$<$," *not*, etc. When backtracking is needed for *if-then-else*, some optimizations of `push` and `pushB` instructions are possible. The `chopBack` can also be used before return or lCall instructions to remove unnecessary choice points (those which did not instantiate anything, so they do not need to be retried) [Sahlin 1986].

Another possibility is to be able to switch the machine to deterministic mode. This would require using a different call instruction (which does not assume link and unlink). In this way, xpPAM would function much like a conventional machine.

4.4 Delays

A delay is handled by using an instruction such as swXVNL or varGoto to detect that a value is uninstantiated – a delay *r,offset* instruction then suspends the predicate by saving a thunk (with all the non-empty registers) on the delay list associated with the variable and executes a *return*.

The calling predicate continues execution until it instantiates a variable which caused a delay. The executing predicate is suspended and the delayed predicate is resumed. The resumed predicate will eventually return, so information about the newly suspended predicate is pushed onto the call stack – the return instruction will use that information to resume it. The net effect is as if the instruction which woke up the delayed predicate had been replaced by link, call *delayed predicate*, unlink and the delayed predicate's return had returned to the unlink instruction.

There may be more than one predicate delayed on a variable, so the oldest such predicate is resumed and all others are pushed onto the execution stack after the suspended predicate. When the delayed predicate returns, the next delayed predicate is resumed and so on until the suspended predicate is reached and resumed. This all happens automatically with the `return` instruction.

When a predicate delays it must also be recorded on the reset stack so that it can be removed from the delay list on backtracking – when a delayed predicate is woken, it is recorded a second time on the reset stack so that backtracking can put it back on the delay list. An optimization similar to tail recursion optimization is performed: if no choice point has been created since the original delay entry was created, both entries are removed from the reset stack (shuffling the stack down if necessary).

Putting all this together, let us trace

```
f(b).
f(c).
?- X ≠ a , X ≠ b , f(X) , X ≠ d.
```

which is compiled to (`X` is in register 0):

```
                        % reg 0 = X
 1     eq    n1,'a'     % reg 1 = 'a'
 2     link
 3     push v0          % save X
 4     call '≠'/2       % call not-equal(X, 'a')
 5     pop   n0         % restore X
 6     unlink
                        % reg 0 = X
 7     eq    n1,'b'     % reg 1 = 'b'
 8     link
 9     push v0          % save X
10     call '≠'/2       % call not-equal(X, 'b')
11     pop   n0         % restore X
12     unlink
13     link
```

```
14     push v0       % save X
15     call f/2      % call f(X)
16     pop  n0       % restore X
17     unlink
18     lCall '≠'/2  % last call not-equal(X, 'd')
```

'≠'/2 must be a separate predicate. If it were expanded in-line, it would cause a delay in the entire predicate, rather than just in the inequality.[14]

```
21      testskip f0,f1 % delays if necessary
22      fail           % reverse success/failure
23      return
```

f/2 is[15]

```
31         pushB v0     % save 1st arg
32         mkCh  else   % make choice point
33         eq    f0,'b'
34         return
35 else:   popB  n0
36         eq    f0,'c'
37         return
```

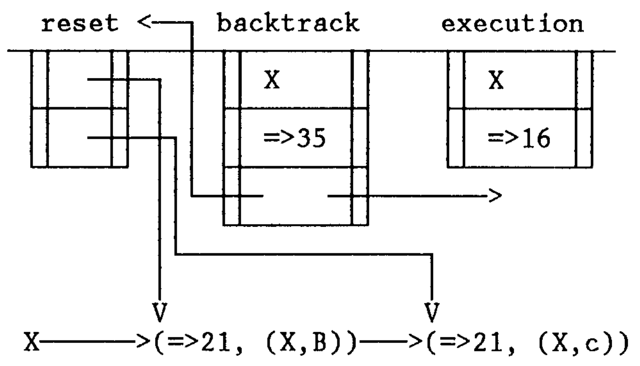
When the first "≠" is executed, it delays because its first argument (X in register 0) is uninstantiated. A delay object is built, containing the values of the registers (X and 'a') and set to redo the testskip instruction (#21). X points to this object. A *return* is simulated. Execution continues to X≠b which also delays

---

14 The current implementation has a built-in for '≠'/2 which delays as necessary. Experience with this built-in lead to the definition of testskip which has *not equals* as a special case.

15 This is not very optimized; a test should be first done for the parameter being instantiated, thereby avoiding the creation of choice point.

– the delay object is added to the list pointed from X. Finally, f(X) is executed which creates a choice point and instantiates X=b. At this point, we have the following situation for the stacks and X.



Note that the simulated returns have popped the execution stack from the two calls to ≠. We haven't reached the return instruction, so the return information hasn't been popped (even if the return had been reached, it still would not be popped because the choice point "protects" it). The notation =>21 means that there is a code pointer to instruction 21.

At this point, the first delay entry pointed from X is tried (it is removed from the list pointed from X but left on the reset stack, marked as having been re-started). It succeeds. The next delay entry is tried but it fails. Backtracking rebuilds the delay entries for X and then jumps to instruction #35. The popB restores the saved register environment (just one register in this case) so that the machine is now in the same state as when the choice point was created (including delay entries). From here on, execution is deterministic because there is no choice point left on the backtrack stack (the reset stack entries will also disappear as the "≠"s are resumed).

To allow *or*-delays, one or more `delayOr` instructions may precede a `delay` instruction. They add information to the delay list entry which is finished by the `delay` instruction. When the `delay` instruction is executed, it places the delayed predicate on the delay lists for all the variables which can cause it to resume.

As another example, consider *append* The code is:

```
% ?- proceed append(A?, B, C).
% ?- proceed append(A, B, C?).
% append(X.A, B, X.C) :- append(A, B, C).
% append([], X, X).

code(append, 3, [
/* 0 */ const([])],
       [swXVNL(0.f, 3.n, 0.n),  % L1 = X . A
        builtin(3),             /* on failure: error/3 */
        goto(l1Var),            /*    var */
        goto(l1Nil),            /*    []  */
     label(l1LstEl),            /*    _._ */
        eqlst(2.f, 3.f, 2.n), % L3 = X . C
                              % arg0: A (already there)
                              % arg1: B (already there)
                              % arg2: c (already there)
        lCallSelf,            % append(A, L2, C)
     label(l1Nil),
        eq(1.f, 2.f),         % L2 = L3
        return,
     label(l1Var),
        nonvarGoto(2, mkch1),
        delayOr(0, 0),        % delay(L0,
        delay(2, 0),          %       L2)
     label(mkch1),
        pushB(0.v), pushB(1.v), pushB(2.v),
        mkCh(mkch2),
        eqlst(0.f, 3.n, 0.n), % L1 = X . A
        goto(l1LstEl),
     label(mkch2),
        popB(2.n), popB(1.n), popB(0.n),
        eq(0.f, 0.c),          % L1 = []
        goto(l1Nil)]).
```

The code from `label(l1Var)` on handles delays. The delay restarts at the beginning so that the `swXVNL` instruction will be re-tried. The code from `label(mkchl)` on sets up for backtracking. Each choice point instantiates the first argument to one of the possibilities (a list element or *nil*) then jumps to the deterministic code for that case.

4.5 Cost of delaying

The current implementation has a slight cost when instantiating a variable: it checks to see if the variable has caused a delay. This can be changed by making a new object type: "uninstantiated variable which caused a delay." As mentioned earlier, adding new object types causes no slowdown because it results in just adding extra cases to a branch table.

Currently, when unification instantiates a variable, a flag is set to indicate that a delayed predicate is eligible for waking up. This means that every `xpPAM` instruction must check to see if it should be preempted. This is potentially expensive. There are three possibilities:

- Only check for woken predicates at expensive instructions, such as *call* or *return*.
- Allow instructions to be halted in the middle and later resumed. This adds significantly to the abstract machine's complexity.

- Do complex unifications by a user-defined "eq" predicate (see section 3.13, "User defined unification" on page 58.). This is probably the best solution because it turns unification into a non-atomic operations which can be delayed at any time by already existing mechanisms.

The overhead for providing delays is quite low: about 1%. Because delays can speed up some predicates by an order of magnitude or so, the cost is worthwhile.

4.6 Weak delays: dynamic reordering of clauses

The ancestor predicate is inefficient if the first argument is uninstantiated:

```
ancestor(Ancestor, Descendent) :-
     parent(Ancestor, Descendent).
ancestor(Ancestor, Descendent) :-
     parent(Ancestor, Z),
     ancestor(Z, Descendent).
?- ancestor(X, george).
```

In the second clause, `parent` with two uninstantiated variables will repeatedly generate all `parent` relations by backtracking. Changing `parent` to delay until both parameters are instantiated would prevent this and give efficient execution, but would also cause `ancestor` to delay permanently if it is called with two uninstantiated variables, for example when computing ancestors beyond grandparent. To handle this, xpPAM allows an associating a "cost," proportionate to the size of the predicate's solution space, with each delay instruction. For example, if there are 100 parent-child relationships, an average of 2 parents per child and 3 children per parent, the code would be:

```
     varGoto    r0, L1
     varGoto    r1, L2
L0:    ... code for both r0 and r1 instantiated.
     return
L1:  varGoto    r1, L3
     delayCost  2       % delay parm0, cost=2
     delay      r0, L1a % resume at next instr
L1a: notvarGoto r0, L0  % parm0 possibly var
       ... code for r0 uninstantiated and r1 instantiated.
L2:  delayCost  3       % delay parm1, cost=3
     delay      r1, L2a
L2a: notvarGoto r1, L0
       ... code for r0 instantiated and r1 uninstantiated.
L3:  delayCost  100     % delay parm1,
     delayOr    r1      % or parm2, cost=100
     delay      r0, L3a
L3a: varGoto    r0, L2
     notvarGoto r1, L0
       ... code for r1 uninstantiated and r1 uninstantiated.
```

As before, the machine resumes clauses when their arguments are sufficiently instantiated. If all predicates are blocked, the least expensive one is resumed. The machine thereby dynamically decides the least expensive way to continue a non-deterministic computation.

Ordinary delays could also be treated something like weak delays with extremely high costs. Usually, when computation halts with some predicates still delayed, the answer is not "yes" or "no" but "don't know." Instead, the delayed predicates could be allowed to resume, to try to get to a definite answer. Unfortunately, this will often lead to infinite loops.

4.7 Delay instructions

delay *op1*, *offset*

The object pointed by *op1* is flagged as having caused a delay (the object must be an uninstantiated variable). A delay object is created (pointed to by the *op1* object) with sufficient information to restart the predicate at *offset*. If a `delayOr` instruction had been executed, its delay object is linked in. Finally, a `return` is simulated.

delayOr *op1*, *offset*

The object pointed by *op1* is flagged as having caused a delay. A delay object is created (pointed to by the *op1* object) with sufficient information to restart the predicate at *offset*. A flag is set so that a subsequent `delay` instruction will link in this delay object. Execution continues at the next instruction.

delayRec *op1, offset*

Like `delay` except that a delay is made if an uninstantiated variable is found anywhere within the operand. The delay test is recursive on compound objects (list elements, etc.).

delayCost *op1, cost*

Like `delay` but with an associated cost. See section 4.6, "Weak delays: dynamic reordering of clauses" on page 81)

mkThunk *op1, op2, #regs*

Creates a thunk in *op1* from *op2* The operand may be either a constant (a code segment) or a thunk. The thunk is constructed from the first *#regs* registers plus any registers specified by the operand (if it is a thunk). For more details, see section 6.4, "Thunks, lazy evaluation and higher order functions" on page 127.

5.0 Compiling Prolog

There are only two qualities in the world:
efficiency and inefficiency ...
— George Bernard Shaw, *John Bull's Other Island, act 4*

XpPAM is designed to make compilation easy. My simple compiler from Prolog to xpPAM is about 600 lines of Prolog code – a full compiler from xpProlog to xpPAM is about twice that, to take care of delays, *if-then-else*, etc. Optimization requires considerably more analysis which is essentially independent of the target machine (5000 or so lines of Prolog[16]).

Compiling from Prolog to xpPAM is somewhat different from compiling to WAM. In particular, allocating "local" and "global" variables – is a problem which does not exist with xpPAM. Some approaches to compiling Prolog to WAM are given in [Debray 1985], [Debray and Warren 1986] and [Van Roy 1984].

16 This number is derived from the compiler which is described later in this section. About 2000 lines of Prolog did most of the optimizations described here; I estimate that the full compiler would be about twice as large, for the same reason that the full xpProlog compiler is about twice as large as the simple Prolog compiler.

Three types of optimization are possible for logic programs:

Clause optimizations include common expression elimination and removing unnecessary register copying. These are fairly straightforward – they are similar to "peephole" optimizations in conventional compilers.

Predicate optimizations mainly consist of detecting determinism and common sequences of goals among clauses; in effect, turning clauses into *if-then-else* form where possible. To do a good job, these must have knowledge of predicates such as *equals* and *not equals.*

Global optimization include flow analysis, mode inferencing, delay inferencing and combining predicates (to avoid the overhead of calling "helper" predicates). These can become quite involved, although they are considerably simpler than similar techniques for conventional programming languages because of the lack of destructive assignment.

The programmer can help the optimizer. As predicate optimization consists partly of turning clauses into *if-then-else* form, the programmer can, if he wishes, use *if-then-elses* from the beginning. However, the programmer's style may use conventional clause form without penalty.

Although some Prolog predicates can be run "backwards" (such as *append* for splitting lists), many cannot. The programmer may indicate this to the compiler. In the absence of full global type inferencing, the compiler must still generate code for dynamically detecting uninstantiated variables but it can save considerably on the amount of code generated and help debugging by generating calls to error instead of backtracking and failing.

There is almost no limit to the amount of optimization that can be done. For example, `link` and `unlink` instructions can be left out if global analysis determines that the predicate is never called in the context of a choice point. However, such a level of optimization is probably not desirable because these instructions are not very expensive and because such optimizations require substantial checking whenever a new predicate is added.

In the following discussion, clause, predicate and global optimizations are described as separate processes. However, they may often be combined.

The compiler is written entirely in Prolog. The raw code for a predicate is stored in an internal form which is either compiled right away or compiled on first use. For example,

```
pred([], A, B.C) :- zot(A).
pred(a, [], []).
```

is stored internally as:

```
[[[const.pred, const.[], name.'A', list.(name.'B').(name.'C')],
   [const.zot, name.'A']],
 [[const.pred, const.a, const.[], const.[]]]]
```

The entire predicate is turned into something like the following, where *predClauses* is the internal form of the clauses (described in the previous paragraph):

```
pred(P1,P2,...,Pn) :-
   compile(predClauses, Name, NArgs, ConstList, Code),
   /* compile instantiates Name, NArgs,ConstList, Code */
   replacePred(pred(P1,P2,...,Pn), Name, NArgs, ConstList, Code)

replacePred(Pred, Code) :-
   addCode(Name, NArgs, ConstList, Code),
   call(Pred). % calls compiled version of Pred
```

Note that the call to *replacePred* is tail recursive. *AddCode* replaces the body of *pred* with the newly compiled code. *ReplacePred* then tail recursively calls that code. This extra step is necessary because otherwise the machine would be attempting to execute code while it was being replaced.

5.1 Unoptimized compiling of a predicate

For simplicity, I will consider only clauses without any imbedded "or"s or *if-then-elses*. Predicates containing "or"s can always be transformed by producing auxiliary predicates. For example:

```
p(A, B) :- q(A, Z), (r(A, B) ; s(Z)), t(A).
```

can be transformed to

```
p(A, B) :- q(A,Z), p2(A, B, Z), t(A).
p2(A, B, _Z)  :- r(A, B). /* _Z is unused */
p2(_A, _B, Z) :- s(Z).    /* _A, _B, are unused */
```

If-then-elses are generated as an optimization and are treated later.

The general form of a compiled predicate is:

```
make choice point
    first clause
modify choice point
    second clause
modify choice point
    third clause
  ...
remove choice point
    last clause
```

If there is only one clause, choice points are not required. If there are two clauses, there are only "make choice point" and "remove choice point" instructions (no "modify choice point"s).

The general compiled form of a clause is

```
unify arguments in head
build arguments for first goal
call first goal
build arguments for second goal
call second goal
  ...
build arguments for last goal
call last goal tail recursively
```

If there are no goals, a "`return`" instruction is generated after the arguments in the head are unified. If the last goal is to itself, a *last-call-self* (in effect, *goto*) instruction is used.

5.2 Basic compiling of a clause

At the beginning of a clause, the first *n* registers are occupied by the *n* arguments to the predicate. All other registers are assumed to be empty. As described in section 3.2, "General registers" on page 32, if there are more arguments than machine registers, the extra arguments must be combined into a structure in the last argument.

Each argument which is a simple variable name is assigned to the appropriate register, except when the name is used elsewhere in the head (this prevents some pathological cases for trace analysis; see section 5.5, "Optimized compiling of a predicate" on page 98). All other names are assigned sequentially from the unused registers. For example, in

```
p(abc, A, Z, B, X.Y, B, X) :- q(xyz, X, X, A.C), r(Z, C).
```

the following register assignments are produced from the head (`_r`*n* is a temporary name for register *n*):

Register	Variable	Comment
0	`_r0`	`abc` is treated as an expression
1	`A`	
2	`Z`	
3	`_r3`	`B` is also used in another argument

Register	Variable	Comment
4	_r4	
5	_r5	
6	_r6	
7	B	
8	X	
9	Y	

These register assignments will change as the goals are processed. C is not used in the head and therefore has no initial register assignment.

The clause is transformed into a series of equalities. for A=(B.C).

```
p(_r0, _r1, _r2, _r3, _r4, _r5, _r6) :-
    _r0 = abc
    _r1 = _r1                   /* A */
    _r2 = _r2                   /* Z */
    _r3 = _r7                   /* B */
    _r4 = _r8 . _r9             /* X.Y */
    _r5 = _r7                   /* B */
    _r6 = _r8                   /* X */
    /* register assignments from here have not yet been determined */
    startCall(4)                /* q has 4 parameters */
    _r0 = xyz
    _r1 = X
    _r2 = X
    _r3 = A . C                 /* _rc = A.C */
    call q/4
    startLastCall(2)
    _r0 = Z
    _r1 = C
    lastCall(r/2)
```

More complex expressions are broken down into individual unification steps (this is sometimes called "partial evaluation." For example, [A,(B.C),D] is handled by

```
_temp1 = (A._temp2)
_temp2 = (_temp3 . _temp4)
_temp3 = (B.C)
_temp4 = (D.[])
```

where the temporaries are also registers. For calls, the order is reversed to build up an argument:

```
_temp1 = (B.C)
_temp2 = (D.[])
_temp3 = (_temp1 . _temp2)
_temp4 = A._temp3
```

Once everything has been assigned, the list is scanned from beginning to end, adding *n* (*new*) every time a variable name is met for the first time; it is then scanned from end to beginning, adding *f* (*free*) every time a variable is met for the last time. Each *startCall* or *startLastCall* clears all the "new" indicators but not the "free" indicators – it gets a list of all the variables needed after it (created during the backward scan and pruned during the forward scan). It is possible to have both "new" and "free" for the same variable; typically this happens when an anonymous variable ("_") is used. (This step is actually combined with the following step, for efficiency.) Variables with neither *f* nor *n* already have a value which is needed later.

```
p(_r0, _r1, _r2, _r3, _r4, _r5, _r6) :-
    f._r0 = abc
    f._r1 = _r1                 /* A */
    f._r2 = _r2                 /* Z */
    f._r3 = n._r7               /* B */
    f._r4 = (n._r8) . (n._r9)  /* X.Y */
    f._r5 = f._r7              /* B */
    f._r6 = _r8                /* X */
    startCall(4, [f.Z, n.C])  /* with list of variables to save */
    n._r0 = xyz
    n._r1 = X
    n._r2 = f.X
    n._r3 = (f.A) . (f.C)      /* A.C (C was "new" at startCall) */
    call(q/4)
    startLastCall(2)
    n._r0 = f.Z
    n._r0 = f.C
    lastCall(r/2).
```

The variables used within calls can now be assigned registers. A call is transformed into a series of pushs, followed by the call, followed by pops into the appropriate registers for the following call. Sometimes, registers must be moved to make room for the parameters for a call. Some equalities can be eliminated or turned into free or new instructions. The example now becomes:

```
p(_r0, _r1, _r2, _r3, _r4, _r5, _r6) :-
    f._r0 = abc
    /* _r1 = _r1 */            /* A */
    /* _r2 = _r2 */            /* Z */
    f._r3 = n._r7              /* B */
    f._r4 = (n._r8) . (n._r9)  /* X.Y */
    f._r5 = f._r7              /* B */
    f._r6 = _r8                /* X */
    link                       /* startCall(4, [f.Z,n.C]) */
    push(f._r2)                /* Z: not used in call */
    push(n._r4)                /* C (new) */
    n._r0 = xyz
    n._r5 = f._r1              /* get A out of the way */
    n._r1 = _r8                /* X */
    n._r2 = f._r8              /* X */
    n._r3 = (f._r5) . (f._r4)  /* A.C */
    call(q/4)
    unlink
    pop(n._r1)                /* C */
    pop(n._r0)                /* Z */
                              /* startLastCall(2) */
    /* n._r0 = f._r0 */       /* Z: already in reg 0 */
    /* n._r1 = f._r1 */       /* C: already in reg 1 */
    lastCall(r/2).
```

Note how the pops get the arguments for a call into the correct registers. The only time that registers must be moved is for the first call in a predicate (for example, A had to moved in the above code). In all cases, pushes and pops will produce the correct effect.

The final stage transforms this into the form used by the built-in *code* predicate by collecting the constants (note that the *code* predicate reverses the order of the register number and its annotation – there is no good reason for this, beyond making the program for processing *code* a little simpler):

```
code(p, 7, /* p/7 */
/* constant 0 */ [const(abc),
/* constant 1 */  const(xyz),
/* constant 2 */  call(q,4),
/* constant 3 */  call(r,2)],
[
    eq(0.f, 0.c),             /* abc */
                              /* A */
                              /* Z */
    eq(3.f, 7.n),             /* B */
    eqlst(4.f, 8.n, 9.n),     /* X.Y */
    eq(5.f, 7.f),             /* B */
    eq(6.f, 8.v),             /* X */
    link,
    push(2.f)                 /* Z: not used in call */
    push(4.n),                /* C (new) */
    eq(0.n, 1.c),             /* arg 0 = xyz */
    eq(5.n, 1.f),             /* get A out of the way */
    eq(1.n, 8.v),             /* arg 1  = X */
    eq(2.n, 8.f),               /* X */
    eqlst(3.n, 5.f, 4.f),       /* A.C */
    call(2.c),                  /* q/4 */
    pop(1.n)                    /* C */
    pop(0.n),                   /* Z */
    unlink,
                                /* Z: already in reg 0 */
                                /* C: already in reg 1 */
    lCall(3.c)                  /* r/2 */
```

If any delay annotation exists (either by "?"s or from *proceeds*), they can be easily incorporated by adding `vargoto`, `nonvargoto` and `delay 0` instructions, as appropriate. These will be discussed in more detail below.

5.3 Optimized compiling of a clause

The above method of compiling a clause can produce some inefficiencies:

- Values may be moved unnecessarily between registers. Above, there was no need to move `B` from register 3 to register 7 when unifying the head of the clause. As another example,
  `foo(A) :- A=B, C=x, B=C.`
  can be optimized to just `foo(x)` (which is actually handled as
  `foo(_r0) :- _r0=x`).

- Common expressions are not noticed. For example,
  `ordered(A.B.C) :- A =< B, ordered(B.C)` is transformed to

```
ordered(_r0) :- _r0 = (A._templ),
                _templ = (B.C),
                A =< B,
                _temp2 = (B.C),
                ordered(_temp2).
```

But _temp2 is the same as _templ and can be eliminated. The compiler code for detecting these common expressions uses uninstantiated variables in an interesting manner. Each expression is replaced by an uninstantiated variable. If that expression is found later, it is replaced by a new temporary and _temp=*expr* is added to the body; if the expression is not found later, the uninstantiated variable is replaced by its original value. It should be noted that the expressions are always very simple because everything has already been broken down to either simple "equals" or "equals list element" form.

- The sequence of instructions

  ```
  new(_rn),
      ...
  eq(_rn,X)
  ```

  can have the "new" removed and the "eq" replaced by "eq(_rn.n,X)."

- Some *goto* or *last call* instructions can be replaced by their target instruction(s).

These optimizations can be done in a straightforward way although they are not all trivial. Sometimes, they can be combined with the other steps, to avoid extra passes over the clause.

5.4 Spilling registers within a clause

The previous sections have assumed that there are enough registers in `xpPAM` to hold all the variables in a clause. Although 30 is a rather large number for predicates written by humans, it is possible that machine-generated predicates could contain many more.

The `push` and `pop` instructions can be used to save and restore registers using the execution stack. The code generation scheme above makes the `link` and `push` instructions for a call as late as possible. If register spilling is needed, the `link` must be made earlier (if there is a `link`) and `push`es must also be made earlier. In this way, some registers may become free and assignable to other variables.

If there are still not enough registers, more drastic action must be taken. Consider compiling

```
p(A.B, C.D) :- q(A.C), r(A.B.C.D).
```

on a machine with only four registers. The code would be:

```
eqlst f0,n2,n3    /* arg 0 = A.B */
link
push  f3          /* save B */
eqlst f1,n3,n0    /* arg 1 = C.D */
push  v2          /* save A */
push  v3          /* save C */
push  f0          /* save D */
eqlst n0,f2,f3    /* arg 0 = A.C */
call  q/1         /* q(A.C) */
pop   n3          /* restore D */
pop   n2          /* restore C */
eqlst n0,f2,f3    /* r0 = C.D */
pop   n2          /* restore A */
pop   n3          /* restore B */
eqlst n0,f0,f3    /* r0 = B.C.D */
eqlst n0,f0,f2    /* r0 = A.B.C.D */
unlink
lCall r/1
```

Clearly, the pushing and popping on the execution stack can become quite complicated. To simplify this, an arbitrary element on the stack can be accessed by the `gete` instruction. Using `gete` the execution stack can be treated much like a stack frame with local variables in a conventional stack machine. Unneeded entries on the stack can be removed by `pop x31` which pops the value and discards it.

5.5 Optimized compiling of a predicate

One of the most important optimizations is to detect potentially deterministic situations and transforming them into *if-then-elses*. If this is not done, unnecessary choice points are created. Not only are these expensive to create and delete but they also result in unnecessary computations on backtracking. Optimization should produce code without any choice points when a predicate is called with all arguments sufficiently instantiated – in such situations, no new information is produced by re-trying the predicate (it would simply succeed again, or fail if there were no other way to succeed).

Detecting *if-then-else* situations requires first putting the clauses into "standard form" and then computing "traces." A trace is a "most general form" for the parameters of a clause – it is used for finding potentially deterministic parts. For example, consider

```
(1)  m([], M, M).
(2)  m(M, [], M).
(3)  m(X.A, Y.B, X.M) :- X=< Y, m(A, Y.B, M).
(4)  m(X.A, Y.B, Y.M) :- X > Y, m(X.A, B, M)
```

The traces are:

```
(1)  [],   _,  _
(2)  _,    [], []
(3)  _._, _._. _._
(4)  _._, _._, _._
```

The traces can be partitioned on the first argument:

```
[]   (1)
_    (2)
_._  (3), (4)
```

Clause 2 can match anything, including [] and _._, so it must be placed in an *alternate* which is a non-deterministic choice. The other clauses can be placed in a *switch* which is deterministic if the switch value is instantiated. This results in

```
alternate
  switch _r0
    case []:
      clause 1
    case X.A:
      alternate
        clause 3
      else
        clause4
      end-alternate
  end-switch
else
  clause 2
end-alternate
```

This can only be done if the compiler can re-arrange the order of the clauses (valid in a pure logic programming language).

We can make *m* deterministic by changing clause 2:

```
(1)  m([], M, M).
(2)  m(X.A, [], X.A).
(3)  m(X.A, Y.B, X.M) :- X=< Y, m(A, Y.B, M).
(4)  m(X.A, Y.B, Y.M) :- X > Y, m(X.A, B, M)
```

The partitioning on the first argument is:

```
[]      (1)
_._     (2), (3), (4)
```

The second set can be further partitioned by the second parameter:

```
[]      (1)
_._     (3), (4)
```

Clauses 3 and 4 can be partitioned because "=<" and ">" are disjoint. This results in

```
switch _r0
  case []:              /* (1) */
  case X.A:             /* (2), (3), (4) */
    switch _r1
      case []:          /* (2) */
      case Y.B:         /* (3), (4) */
        if X =< Y
          clause 3
        else
          clause 4
    end-switch
end-switch
```

Some care must be taken in analyzing traces. A case may not contain a variable which was "bound" by a earlier case. For example, naïve analysis of

```
foo(X.A, X.B) :- goal1(X).
foo(X.A, Y.A) :- goal2(X,Y).
```

would produce

```
switch _r0
  case X.A:
    switch _r1
      case X.B:
        alternate
          goal1(X)
        else
          goal2(X,X) /* wrong! */
        end-alternate
    end-switch
end-switch
```

instead of

```
switch _r0
  case X.A:
    switch _r1
      case _t1.B:
        alternate
          _t1 = X, goal1(_t1)
        else
          goal2(X, _t1)
        end-alternate
    end-switch
end-switch
```

This could be taken care of by an analysis of "bound" variables when generating the cases, but the analysis is rather complicated. Instead, the compiler relies on the clause compiler guaranteeing that each variable appears in at most one functor. This can result in some unnecessary register moving. In practice, the unnecessary moves (which are cheap) do not happen very often; they can be removed by a second pass of the peephole optimizer.

The traces are used to turn the predicate into a tree of *alternates* (non-deterministic) and *switches* (deterministic). This is done repetitively, one layer of the parameter at a time (for example, A.B.C has three layers, so the process could be repeated up to three times). At each stage, the parameter which produces the largest number of partitions is chosen — if there are any delay ("?") notations then only the parameters so marked are considered because all other parameters are assumed to be called with uninstantiated variables.

Once the predicate is transformed entirely into a tree of *alternates* and *switches*, it is optimized so that *alternates* with only one choice (no *else*) are replaced by their single choice.

The final stage transforms the tree of *alternates*, *switches* and *tests* into xpPAM machine code. *Alternate* is transformed to

```
        mkCh      label1
        pushB     0
          ...
        pushB     n
   ... first choice ...
label1: popBKeep n
          ...
        popBKeep 0
        mkCh      label2
label2:    ... second choice ...
labeln: popB      n
          ...
        popB      0
   ... last choice ...
```

A *switch* is a bit more complex than an *alternate* because it must provide for both a deterministic and a nondeterministic case. It uses the swXVNL and eqskip instruction. For example

```
switch _r0
  case []:  clause 1
  case 'a': clause 2
  case 'b': clause 3
  case 'c': clause 4
end-switch
```

becomes

```
swXVNL 0
    goto atom /* it's not a list element, nil or variable */
    goto var  /* it's a variable */
    goto nil  /* it's [] */
    fail      /* it's a _._ */
nil:
    clause 1   /* this will contain a "return" or "last call" */
atom:
    caseGoto 0
    case(fail) /* not [],'a', 'b' or 'c' */
      case 'a',c2
      case 'b',c3
      case 'c',c4
    case-end
c1: clause2   /* each clause ends with a "return" or "last call" */
c2: clause3
c3: clause4
            /* non-deterministic case ... */
var:
    make choice alt1
    eq     0,[]
    goto   nil
alt1:
    modify choice alt3
    eq     0,'b'
    goto   c2
alt2:
    modify choice alt3
    eq     0,'b'
    goto   c3
alt3:
    remove choice
    eq     0,'c'
    goto   c4.
```

where *make choice*, *modify choice* and *remove choice* are the code sequences as given for *alternate* above. An alternative to the above code would be

```
caseGoto 0
case(var)
case(fail) /* not [],'a', 'b' or 'c' */
  case [],nil
  case 'a',c2
  case 'b',c3
  case 'c',c4
case-end
...
```

The first form would be preferred if "[ ]" were more common; the second form if less common. (Incidentally, the `caseGoto` could have been synthesized from `eqskip`s.)

5.6 Shallow backtracking

The above compilation of *if-then-else* will execute slower than similar code on a conventional machine. Better *if-then-else* code is:

- Test for uninstantiated variables.
- Push all active registers onto the backtrack stack.
- Push machine state and "else" address onto the backtrack stack.
- Create choice point.
- Call the test predicate
- On success: remove the backtrack entries and cut the backtrack stack using `rmChCut`.
- On failure: pop the backtrack stack entries.

Note that no registers are pushed onto the execution stack: the backtrack stack is used for them. Thus, the only overhead in an *if-then-else* is in testing for uninstantiated variables and in creating the choice point.

The test for uninstantiated variables can be removed if we know that the test will not instantiate anything (this can be detected by a method similar to that which automatically generates *proceed* annotations). The test predicate itself can also be transformed to have its own delays.

5.7 Global optimizations

Very often a number of predicates are used together as a module. For example, *quick-sort* typically has a *partition* predicate which is used only by *quick-sort*. When the compiler knows exactly how a predicate is called, further optimizations are possible:

- If the helper predicate is not recursive, it can be substituted into the body of the calling predicate (this is especially useful for built-in predicates such as arithmetic and comparison). For example:

  ```
  p(A, B) :- q(B, X), r(X, A).
  q(X1, X2) :- s(X1), t(X2).
  ```

  can be replaced by:

  ```
  p(A, B) :- s(B), t(X), r(X, A).
  ```

However, some care must be taken. If *q* has any delays in it, this substitution can not be done because the delay would take effect for predicate *p* instead of just *q*.

- The assignment of registers to the helper predicates can be changed to be more efficient for the calling predicate. For example, in:

  ```
  p(A, B, C, D) :- q(a, A, B, C, D).
  ```

  the arguments to *q* can be passed in registers 4, 0, 1, 2, 3 so that there is no need to move *A* from register 0 to register 1, *B* from register 1 to register 2, etc.

- Delays can be propagated outwards. For example, in:

  ```
  p(X, Y, Z) :- X <  Y, q(Z).
  p(X, Y, Z) :- X >= Y, r(Z).
  ```

  Delay tests could be done at the beginning of *p* (indicated by `?-proceed p(?,?,-)`) so that a special non-delaying version of the inequality tests could be used. In some cases, this also allows removing tests for uninstantiated variables from *if-then-elses*. However, care must be taken so that possible solutions are not eliminated — in this example, the optimization could only be done if predicates *q* and *r* do not cause their argument to become instantiated.

- Some cases of garbage collection can be done explicitly. See [Kluzniak 1987] and [Bruynooghe 1986].

5.8 Compiling a predicate with delays

Predicates with delays only slightly complicate the compiler. As mentioned above, the delay annotations are used to restrict which parameters are used for computing traces. The delay annotations can be carried through to the final code generation stage. At this point, they can occur in the following situations:

- As part of an `eq`, `eqskip`, `eqlst` or `caseGoto` instruction. `NonvarGoto` and `delay` instructions are added. For example `eqlst v0?,v1?,n2` would become

```
          nonvarGoto v0, label1
          delay      v0, label1
label1: nonvarGoto v1, label2
          delay      v1, label2
label2: eqlst      f0, n1, n2
```

- As part of a `swXVNL` instruction. If it is the first operand, the uninstantiated variable situation is already detected. Thus, `swXVNL f0?,n1,n2` would become

```
switch: swXVNL     f0, n1, n2
        goto atom
        goto var
        goto nil
        goto lst
var:    delay      v0, switch
```

Or-delays make things a bit more complicated. Consider *append*:

```
?- proceed append(a?, b, c).
?- proceed append(a, b, c?).
append([], X, X).
append(X.A, B, X.C) :- append(A, B, C).
```

If only the first *proceed* declaration were given, this would be

```
switch _r0?
  case []:
    _r1 = _r2
  case X.A:
    _r2 = X.C, append(A, _r1, C)
end-switch
```

However, if the first argument is uninstantiated, execution may still proceed if the third argument is instantiated. To show this, *switch* may have a *var* case:

```
switch _r0
  case []:
    _r1 = _r1
  case X.A:
    _r2 = X.C, append(A, _r1, C)
  var:
    alternate _r2? : delay-or _r0, _r2
      _r0 = [], _r1 = _r2
    else
      _r0 = X.A, _r2 = X.C, append(A, _r1, C)
    end-alternate
end-switch
```

The `delay-or` means that if none of the designated values are uninstantiated, `delayOrs` must be generated. The code is

```
    /* register 2 is already known to point at an
       uninstantiated variable. */
    nonvarGoto v2, label1
    delayOr    v0, 0
    delay      v2, 0
label1: ...
```

This causes *append* to delay until either what is in register 0 or what is in register 2 becomes instantiated. Execution will re-start at the beginning (offset 0) because when *append* is resumed, there is no way of knowing which variable became instantiated and caused *append* to resume.

5.9 Optimized compiling to conventional machine code

The xpPAM code can be interpreted or it can be translated to conventional machine code. The simplest translation would simply use the separate cases in the interpreter as code templates, removing the interpreter loop. This typically gives an execution speed-up of 2 − 3 times at the cost of increasing the code size by a factor of 10 or more.

Significant execution speed-ups are possible by using machine code (assembler) instead of a higher level language such as C or Pascal. This is because assembler provides more possibilities of exploiting the machine's registers. The machine registers can be used for:

- pointer to top of execution stack.
- pointer to next free object.
- the first *n* registers of xpPAM.

This can easily double the speed of the generated code. Note that registers are not used to hold information that is used only on backtracking; xpPAM is optimized for deterministic computations.

Assigning the **xpPAM** registers to machine registers has an additional advantage. When **xpPAM** is interpreted, the operands must be used as indexes into the vector of "registers." Machine code references these directly, even if they have to be kept in slower main storage because of insufficient hardware registers. To give a feel for the speed-ups possible, I changed the implementation to store an offset into the register vector instead of an index. This avoided some shift instructions, giving about 10% speed-up on an MC68000.

Some other optimizations are possible when compiling to machine code. These are typical of optimizers for conventional languages, such as "peephole" optimizations for removing redundant load/stores. Some additional operand annotation, such as *r* to indicate that de-referencing is not needed, can help optimize the code.

Careful tuning of the machine code can produce dramatic speed-ups. In an experiment, the following were observed for naïve reverse:

1.4 KLIPS with a simple clause interpreter.
5 KLIPS with a simple abstract machine interpreter.
10 – 20 KLIPS with more sophisticated abstract machine interpreters.
20 – 40 KLIPS with translation to C.
over 100 KLIPS with translation to native code.

The generated machine code must be position independent because it resides in the *xArea* extension area and may be moved around by the compactor. Some machines have separate instruction and data spaces; for these, separate *xAreas* would have to be provided (these areas could still require compaction, because of incremental re-compilation).

The design of special purpose computers for logic programming is discussed elsewhere, for example [Tick and Warren 1984] and [Mills 1986]. Such machines could probably directly execute xpPAM code significantly faster than could conventional machines, using the same technology. Major speed-ups are:

- Parallel execution of parts of instructions (e.g., tag decoding).
- Maintaining shadow registers and caches which are better oriented to xpPAM's referencing style.

5.10 Speeding up deterministic predicates – modes and types

A type can be considered to be a predicate which is evaluated at compile time and whose failure causes a compilation error rather than a run-time error (or, worse, an unexpected failure in a predicate which should always succeed). Such a predicate either checks an already instantiated variable or imposes a constraint on an uninstantiated variable. This can be implemented in one of three ways:

- The type predicate simply delays until the argument becomes sufficiently instantiated.
- A type flag is set for the variable so that unification will fail if an attempt is made to unify the variable with something of the wrong type.
- Compile-time analysis can propagate the type information outward and eliminate tests. For example, in the following, type propagation could

discover that both the argument for *q* and *Hd* in p must be integers, so no test is needed if p is called with a list of integers:

```
p([]).
p(Hd.Tl) :- q(Hd), p(Tl).
q(X) :- integer(X), ...
```

Typing predicates are useful debugging tools. Instead of just failing with the wrong type argument, they cause a run-time or compile-time error. Typing predicates can be extended to work for more complex types. For example, lists can be handled by:[17]

```
/* general lists: */
list([]).
list(Hd.Tl) :- list(Tl).

/* list of a specific type: */
list(Type, []).
list(Type, Hd.Tl) :- Type(Hd), list(Type, Tl).
```

These predicates are included in the program just like ordinary predicates. The compiler can be told which predicates are for typing – they are executed at compile time and used for type inferencing. The basic built-in types include *integer*, *number* and *string*.

---

[17] Note the use of **xpProlog**'s notation in *Type(Hd)*. In most Prologs, this would be written:

```
Test =.. [Type, Hd], call(Test)
```

(where `call(Test)` could be replaced by `Test`).

As an example, here is *intMember* which is like *member* but which only works on lists of integers:

```
?- mode intMember(X, List) :- integer(X), listOf(integer, List).
intMember(X, X._).
intMember(X, _.Rest) :- intMember(Rest).
```

This can be considered as an abbreviation of

```
intMember(X, List) :- integer(X), listOf(integer, List),
                      intMember2(X, List).
intMember2(X, X.Rest).
intMember2(X, Y.Rest) :- intMember(X, Rest).
```

The optimizer could transform this to

```
intMember(X, List) :- integer(X), intMember2(X, List).
intMember2(X, Y.Rest) :- /* integer(Y), */ X=Y, listOf(integer,Rest).
intMember2(X, Y.Rest) :- /* integer(Y), */ intMember2(X, Rest).
```

Here, using type predicates is less efficient than leaving them out. Inferring the types of many predicates is not very difficult – the algorithm is similar to the algorithm for detecting where delays should be added to code (as given in [Naish 1985b]). Thus, the mode of *member* can be inferred as

```
?- mode member(X,List) :- list(List).
```

There is no way to infer that it should be

```
?- mode member(X,List) :- typeOf(X,Type), listOf(Type,List)
```

In fact, this would be over-specifying the type because it would disallow the query `member(1,[1,a])`. Nevertheless, there may be advantages in specifying the stronger type declaration of *member* because more efficient machine code might be generated if a more complex abstract machine model were used which allowed both tagged and untagged objects – in xpPAM, when testing for equality, the tags are first compared, then the values; comparing only the values would be more efficient.

Increases in efficiency could also be produced by input-output mode declarations and inferences. These are in many ways similar to type predicates. Significant speed-ups are possible, as shown by the example in section 2.0, "Fast append on a conventional machine" on page 15.

Besides increasing efficiency, mode and type declarations and inferences help to show the correctness of **xpProlog** programs. Conventional Prolog is much like the C language where type mismatches produce wrong results (Prolog produces unexpected failures); **xpProlog** could be more like Pascal where such errors are detected at compile time.

6.0 Compiling a functional language to the logic engine

... the Form remains, the Function never dies.
— William Wordsworth, *The River Duddon*

xpPAM can be easily adapted as an efficient target for a functional programming language. The delay mechanism, which allows coroutining and sound negation, is sufficient for all common functional programming constructs such as lazy evaluation and handling functions as first class objects. The resulting machine is about as efficient as a machine designed solely for functional programming.

6.1 Introduction

Much has been written about combining functional programming and logic programming. I will only briefly discuss how the two paradigms of logic programming and functional programming should be combined; the reader should see [DeGroot and Lindstrom 1986] which also has extensive bibliographies. Instead, I will concentrate on how xpPAM can efficiently implement functional programming constructs.

The proposals for combining the two paradigms have been roughly classified in [Bosco and Giovanetti 1986]:

functional plus logic: invertibility, nondeterminism and unification are added to a higher order functional language

logic plus functional: first-order functions are added to a first-order logic with an equality theory

Which is best is a matter of debate. Since a logic machine already deals with unification, nondeterminism and invertibility, I have taken the logic plus functional approach. XpPAM can be used to answer one of the open problems cited by [Bosco and Giovanetti 1986]: the kind of computational mechanism necessary to execute functional and logical programming constructs.

6.2 Functional Programming background

For illustrative purposes, I will use some of the syntax of SASL and HASL [Turner, 1979] [Abramson 1984].

In this discussion, I will assume functional programming with no side affects. That is, when I use the term "LISP," I really mean "pure LISP," not using features such as RPLACD.

The *m* clauses of an *n*-ary function *f* are defined:

$f\ a_{11}\ a_{12}\ \dots\ a_{1n} = expr_1$

$f\ a_{21}\ a_{22}\ \dots\ a_{2n} = expr_2$

$\dots$

$f\ a_{m1}\ a_{m2}\ \dots\ a_{mn} = expr_m$

Expressions may be written as:

expr where { definitions }

The definitions may include function definitions, as well as definitions of the form:

$x = 1{:}x$ /* [1, 1, 1, ...] – ":" is the "cons" operator */

$[a,b,c] = [1,2,3]$

etc.

A definition such as: $s\,f\,g\,x\ =\ f\ x\,(g\,x)$ can be thought of as a "syntactically sugared" version of the lambda expression: $s\ =\ \lambda f.\,\lambda g.\,\lambda x.f\ x\,(g\,x)$.

Some functional programming languages have only single argument functions. The single argument and single result, however, may themselves be single argument functions. That is, if f is defined as an n-ary function, it is implemented in such a way that in an evaluation of

$f\ arg_1\,arg_2\ \ldots\ arg_n$

the implicit evaluation is

$(\ \ldots\ (f\ arg_1)\,arg_2)\ \ldots\ arg_n)$

where the result of $(f\ arg_1)$ is a function of a single argument, which can then be applied to arg_2, etc. SASL and HASL treat n-ary functions this way; LISP and Scheme do not.

6.2.1 Normal and applicative order evaluation

Normal order applies the leftmost reduction (evaluation) repeatedly until the normal form (or sometimes just head normal form) of the expression is produced. This means that an argument will never be evaluated if it is not needed within a function body. For example, if $x+1$ is passed as an argument, it will not be evaluated until it occurs in the context of an operator which requires its value, such as an equality test. On Turner's combinator machine, the $x+1$ is replaced by its value so that any subsequent use of the argument obtains the value directly.

Applicative order evaluates the arguments before evaluating the function body. This is the usual way with LISP and Scheme, although they may delay evaluation by having the caller package arguments in lambda expressions (and, in the case of LISP, the called function must be prepared for such arguments).

Both applicative and normal order evaluation produce the same answers if they terminate. However, it is possible that applicative order may do unnecessary computations which may go into an infinite loop. A compromise between the two is "evaluation by need" where arguments are not evaluated until their values are needed – that is, a conventional machine architecture is used but argument evaluation is delayed as late as possible. In evaluation by need, the value of an argument is saved so that it is not re-evaluated if it is used later in the function.

6.2.2 Lazy and eager evaluation.

Lazy evaluation delays evaluations until they are needed, whereas eager evaluation proceeds even though values might not be needed. One effect of lazy evaluation is to permit programming with "infinite" structures such as:

$x = 1 : x$

(an infinite list of 1's: [1, 1, 1, ...]). With lazy evaluation, an expression such as "*hd x*" yields the value 1, the tail of the list not being evaluated. SASL and HASL provide lazy evaluation; LISP, generally, does not, but permits the construction of mechanisms for lazy evaluation.

6.2.3 Lexical and dynamic scoping (deep and shallow binding)

In a language where functions are first class objects, a function must be applied in the correct environment, binding free variables in the function body. The data structure consisting of the function body and an environment is called a closure or a thunk [Ingerman 1961]. In pure functional programming languages, the closure is formed at definition time: this is called lexical scoping or deep binding. Other less pure functional languages form a closure at evaluation time: this is known as dynamic scoping or shallow binding. Pure functional programming requires lexical scoping and is used in such languages as SASL, HASL and Scheme. Many varieties of LISP, however, use dynamic scoping which leads to serious semantic problems, chiefly a kind of destructive assignment.

6.2.4 Mechanical evaluation of functional programming constructs

The earliest programming language with a strongly functional flavour was McCarthy's LISP. An evaluator (interpreter) for a functional subset of LISP was defined in LISP. LISP, however, was not purely functional, allowing destructive assignment, *go tos*, etc. Several years after LISP was introduced, [Landin 1966] described an interpreter for Church's lambda notation. The SECD machine adopts the strategy of applying functions to evaluated arguments and is suitable for programming languages using applicative order. Some lambda expressions, however, can be evaluated only with normal order evaluation; applicative order would result in non-terminating computations.

The "procrastinating" SECD machine, a modification of the original SECD machine, allows the postponement of evaluation of arguments until their values become necessary. In a purely functional setting, this is equivalent to normal order evaluation. The evaluation of an expression can be carried out once, and this value saved if the value of the expression is needed again – the above-mentioned evaluation by need (see [Burge 1975]).

Another major technique for evaluating functional programming constructs was introduced by Turner in an implementation of SASL. The primitive operations of SASL are application of a function to a single argument, and the pairing or construction of lists. In this technique, variables are removed from lambda expressions yielding expressions containing only combinators and global names referring either to library or user-defined functions (from which variables of course have been removed). Evaluation of an expression then is accomplished by a combinator machine: a machine whose instructions correspond to the three fundamental combinators *S*, *K* and *I* and additional combinators which are not strictly necessary but are introduced to reduce the size of the generated combinator machine code. The leftmost possible reduction is performed to yield an expression's head normal form. Normal order evaluation and "lazy" evaluation of lists fall out from this and the definition of the combinator expressions. Furthermore, SASL introduced a limited kind of unification, extended later in Abramson's HASL.

6.2.5 Functional programming via logic programming.

Functional programming languages may be evaluated by logic programs either by interpretation of functional expressions or by compilation of functional programs to logic program goals to be solved. The former technique was used by [Abramson 1984] to specify HASL (an extension of Turner's SASL) in Prolog; the latter technique was used by [Bosco and Giovanetti 1986] to show how various functional programming constructs could be represented as Prolog goals to be solved. Other aspects of the implementation of functional programming extensions of logic programming may be found in [DeGroot and Lindstrom 1986].

Since functional programming constructs can be compiled to Prolog, and since Prolog itself may be compiled, it is possible to stop at this stage and consider the problem of implementing functional programming constructs in a logic programming language as being solved. However, more efficient handling of functional programming constructs is still possible if one compiles them not into Prolog but directly into code for a logic engine. In the remainder of this section we discuss this, in the process gaining about the same efficiency as would be possible by compiling functional programming constructs to an abstract machine designed specifically for functional program evaluations.

The xpPAM's design is a better target for a functional language than is the Warren Abstract Machine (WAM). The principal differences between xpPAM and WAM are:

- Registers contain only **pointers** to objects, not the objects themselves.
- All objects are kept on the heap (like WAM's "global" stack, but not popped on backtracking).
- The unification instructions are simpler (for example, there is no distinction between "local" and "global" variables).
- The execution stack (WAM's "local" stack) contains only call/return information.
- There is no "environment" for a predicate – push and pop instructions are used to save values across calls.

6.3 Sample code

Here is how a function *p* which produces a new list by applying the function *q* to each element of a list may be written in SASL or HASL:

p [] = []
p (Hd:Tl) = (q Hd) : (p Tl)

In Prolog, this is:

```
p([], []) :- !.
p(Hd.Tl, HdX.TlX) :-
    q(Hd, HdX), !, p(Tl, TlX).
```

The *cuts* ("!") are necessary to make this deterministic (xpProlog's *if-then-else* could be used to eliminate these). In general, an *n*-ary function can be turned into an (*n*+1)-ary predicate by adding one parameter to hold the result. This extra parameter must always be initialized as an uninstantiated logical variable before a call and the called predicate must always instantiate it before returning.

Here is xpPAM code (comparable WAM code is in section 7.1, "Comparison with the Warren Abstract Machine instructions" on page 138):

```
  swXVNL    f0,n0,n2 % switch on parm0
  builtin   "error"  % invalid parm
  builtin   "error"  % can't be variable
  goto      nil      % parm0=[]
lst:                 % parm0=Hd.Tl
  eqlst     f1,n1,n3 % parm1:=HdX.TlX (can't fail)
  link
  push      f2       % save Tl
  push      f3       % save TlX
  call      q/2      % q(Hd,HdX): regs 0, 1
                     % are already set
  pop       n1       % arg1:=restore TlX
  pop       n0       % arg0:=restore Tl
  unlink
  lCallSelf          % p(Tl,TlX)
nil:
  eq        f1,[]    % result:=[]
  return
```

The `lCallSelf` instruction has the same meaning as `goto 0` (the different opcode helps in debugging). This is a tail recursive call – recursion has been turned into iteration.

For comparison, here is the pure function version of the function *p* which applies *q* to each element of a list. The result is returned in register 1 which is not pre-initialized to be a logical variable (actually, register 0 should be used to make higher order functions easier to implement – doing this would require more machine instructions for moving values into the correct registers). The `link` and `unlink` instructions are not needed for purely deterministic computations (if used with a slightly different *call* instruction).

```
  swXVNL   f0,n0,n2 % switch on parm0
  builtin  "error"  % invalid parm
  builtin  "error"  % can*t be variable
  goto     nil      % parm0=[ ]
lst:                % parm0=Hd.Tl
  link
  push     f2       % save Tl
                    % arg0: already set
  call     q/1      % reg1:=q(Hd)
  pop      n0       % restore Tl
  unlink
  link
  push     f1       % save q(Hd)
  call     p/1      % r1:=p(Tl)
  pop      n2       % restore q(Hd)
  unlink
  eqlst    n0,f2,f1 % result:=q(Hd).p(Tl) (can't fail)
  return
nil:
  eq       n1,[ ]   % result:=[ ]
 return
```

This code is one instruction longer than the code which uses logical variables (because of the return instruction). Also, the functional code is not tail recursive because of the "cons," so much more stack is needed. The functional version builds the return value more efficiently using `eqlst n0,f2,f1` – the predicate version does it by `eqlst f1,n1,n3` and then filling in the head and tail. Thus, the tail recursion optimization in the predicate has a price: slightly slower construction of the result and one extra level of indirection (using "reference" objects). It also violates the xpPAM convention that all registers are empty when a `return` is executed because it leaves one register containing a value.

Typically, using predicates instead of functions results in about the same number of machine instructions. The slight amount of extra execution time required by using logical variables is more than compensated by the better opportunities for detecting tail recursion optimization (TRO). Without loss of generality, we can use deterministic predicates instead of functions. When we say that a predicate returns a value we mean that the last parameter gets instantiated to the value returned by the equivalent function.

So far, we have treated functions as if they were compiled into Prolog. This can introduce large inefficiencies because of the more general nature of unification and the possibility of backtracking. The `eqskip` instruction is used to avoid creating choice points. For example:

p [] = []

p ('a' : rst) = 'x' : (p rst)

p ('b' : rst) = 'y' : (p rst)

is compiled to:

```
  swXVNL    f0,n2,n0  % switch on parm0
  builtin   "error"   % invalid parm
  builtin   "error"   % can*t be variable
  goto      nil       % parm0=[]
lst:                  % parm0=hd.rst
  eqskip    f2,'a'    % test hd = 'a'
  goto      else
  eqlst     f1,'x',n1 % result := 'x' :
  lCallSelf           %            p(rst)
else:
  eqskip    f2,'b'    % test hd = 'b'
  builtin   "error"   % else: invalid parm
  eqlst     f1,'y',n1 % result := 'y' :
  lCallSelf           %            p(rst)
nil:
  eq        f1,[]     % result := []
  return
```

A boolean function returns either true or false, so calling a boolean function and testing the result can be done in a similar fashion – there is no need to create choice points.

6.4 Thunks, lazy evaluation and higher order functions

A thunk contains the address of an n-ary predicate and the values of the first m arguments ($m \leq n$). The thunk can be considered as an $(n-m)$-ary predicate. When the thunk is called, the first $n-m$ registers are moved into registers m through $n-1$ and registers 0 through $(n-m)-1$ are loaded with the values in the thunk.

In applicative order, the code for `plus(1,2,Z)` is

```
eq    n0,'1'
eq    n1,'2'
link
push  n2            % Z
call  'plus/3'
pop   nZ            % Z
unlink
```

In normal order, the code is transformed to compute

z where {

  $z = p1(2)$,

  $p1 = pluss(1)$,

  $pluss = \lambda x \lambda y \ \{x+y\} \ \}$

Plus 1 2 is reduced to *p1 2* (with $p1(x) = \lambda x\{x+1\}$) and finally to *3*.

On **xpPAM**, we compile the functional expression
z=p1(2) where p1=pluss(1)
to the predicate calls
`?- pluss(1,P1), P1(2,Z).`
which then becomes:

```
  eq    f0,'1'         % argument
  link
  push  n1             % P1
  call  'pluss/2'
  pop   n2             % P1
  unlink
  eq    n0,'2'         % argument
  link
  push  n1             % Z
  callThunk 'f2/2'     % call P1(2,Z)
  pop   nZ             % Z
  unlink
```

`Pluss` is a 1-ary function (2-ary predicate):

```
pluss(X,Z) :- thunk(plus(X)/2,Z).
```

which is compiled to:

```
  mkThunk f1,plus/2,1  % Z:=thunk(...)
  return
```

`MkThunk f1,plus/2,1` means that register 1 is unified to a thunk pointing to *plus/2*, the first argument already having been set in register 0. `MkThunk` is like a `call`, so it frees all the arguments (here, register 0 is marked as empty after its value is saved in the thunk). The third operand to `mkThunk` may be a register so that we can make a thunk from a thunk. `MkThunk` turns atomic objects into thunks by using the "=" predicate (defined "`X=X`").

Whenever an instruction has a thunk as an argument and it requires the value, the thunk is "woken up" and evaluated. The non-empty registers are put into a new thunk (pointing to the current instruction) which is pushed onto the execution stack. This suspends the currently executing predicate (unification is repeatable, so we do not need to store any other information to aid in restarting the suspended instruction). The registers are then loaded from the woken thunk and execution proceeds within it until its last *return* instruction is reached. Normally, when a *return* is executed, the top element on the stack is a code segment but in this case, it is a thunk so the registers are restored from the thunk (recall that all the registers must be empty before a *return*) and execution resumes where it was earlier suspended. This is very similar to how coroutining predicates are suspended and resumed.

When a thunk is evaluated, its result may be another thunk. Therefore, equality instructions may repeatedly wake up thunks until they finally get a value. This is handled automatically because when a thunk returns, it restarts the equality instruction (no state information about the instruction needs to be saved).

Because a thunk saves the registers, pure normal order execution does not need to push and pop registers on the execution stack. But the `mkThunk` and `callThunk` instructions are quite expensive, so using thunks and normal order evaluation is less efficient than applicative order evaluation.

A simple example of delayed evaluation is:

intsFrom m i = m : (intsFrom (m+i) i)

which returns a list of every *i*th integer starting at *m*.

We can sum all the even integers up to *n* by:

```
sumLim (x : r) n = if x > n
                      then 0
                      else x + (sumLim r n)
sumEven n = sumLim (intsFrom 2 2) n
```

At each step of *sumLim*, a new list element (*x:r*) is required. This wakes up the *intsFrom* thunk which makes a list element containing the next number and a thunk for generating the next list element. Eventually *sumLim* reaches the limit *n* and no more elements are needed.

A naïve translation to producer-consumer coroutines is:

```
intsFrom(M, I, M.L) :- M2 is M + I,
                        intsFrom(M2, I, L).
sumLim(X?.R, N, 0) :- X > N.
sumLim(X?.R, N, S) :- sumLim(R, N, S2),
                       S is X + S.
sumEven(N, S) :- sumLim(L, N, S),
                 intsFrom(2, 2, L).
```

where the "?" notations mark where the predicate must delay until the value is instantiated. A delay is implemented by using a `swXVNL` or `varGoto` instruction to detect that a value is uninstantiated – a `delay` instruction suspends the predicate by saving a thunk (with all the non-empty registers) on the delay list associated with the variable and then executing a *return*. When the variable becomes instantiated, all associated delayed predicates are made eligible for resumption (the current predicate is suspended and all the delayed predicates are pushed onto the execution stack except for the oldest which is resumed).

6.4.1 Lazy thunks vs. delayed predicates

The above program has a subtle error. `sumLim` is started and immediately delays on `L`. `IntsFrom` is then entered – it instantiates the first element of `L`. This wakens `sumLim` (and suspends `intsFrom`) which calls itself and then delays on the next element of `L`. Eventually, `sumLim` terminates but `intsFrom` continues generating elements in the list even though they are not needed. Here is a correct version which uses `intsBetween` to generate a finite list:

```
intsBetween(M, N, I, M.L) :- M ≤ N,
                             M2 is M + I,
                             intsBetween(M2, N, I, L).
intsBetween(M, N, I, []) :- M > N.
sum([]?, 0).
sum(X?.R, S) :- sum(R S2), S is X + S.
sumEven(N, S) :- sum(L, S),
                 intsBetween(2, N, 2, L).
```

This should not be taken to mean that functional thunks are more powerful than delayed predicates. Sometimes, delayed predicates are easier to use because they allow more than one predicate to delay on a single variable. Predicates also can backtrack.

The main difference between the two concepts is in how they handle an "infinite" list. For the computation to terminate, the list must be made finite. Thunks do this by eventually leaving the tail uncomputed; delayed predicates eventually instantiate the tail to nil. In both cases, the list need not actually exist (it is, after all, just a communication channel) – reference counting (if used) ensures that only the current element exists, all other elements being deallocated as soon as they are finished with.

6.5 Equality: "is" and "="

Standard Prolog has a simple syntactic equality theory given by the predicate "=" (defined "X=X"). Another kind of equality is provided by the built-in predicate "is" (":=" in Waterloo and IBM Prologs). This can be considered to be defined:

```
X is Y :- atomic(X), X=Y.
X is F :- F =.. Fname.FArgs,
          isArgs(FArgs, FAx), /* evaluate the args */
          append(Fname.FAx,[X],FL2),
          F2 =.. FL2,
          call(F2).
isArgs([], []).
isArgs(H.T, H2.T2) :- H2 is H,
                      isArgs(T, T2).
+(X,Y,Z) :- {Z:=X+Y}. % built-in predicate
-(X,Y,Z) :- {Z:=X-Y}.
    ...
```

The predicates "is," "+," "-," etc. delay if any of their parameters are not sufficiently instantiated.

The first clause is a slight extension of the conventional definition (removing the "only numbers" restriction). The second clause expects the right-hand parameter to be of the form $F(A_1, A_2, \ldots, A_n)$: it evaluates arguments A_1 through A_n (recursively using "is") to produce B_1 through B_n, then computes X by calling $F(B_1, B_2, \ldots, B_n)$. If we assume that "call," "=.." "+" etc. are defined by an infinite number of rules, "is" is definable in first-order logic.

This definition of "is" gives a kind of semantic equality. The more convenient notation p({F}) means Fv is F, p(Fv) ({F} is pronounced "evaluate F"). Using this for factorial:

```
f(0, 1).
f(N, {N * f({N-1})}.
```

which is an abbreviation for:

```
f(0, 1).
f(N, NF) :- NF is N*F, Nsub is N-1,
            f(Nsub, F).
```

This definition depends on arithmetic predicates delaying when their arguments are insufficiently instantiated. Although tail recursive, it is much less efficient than the non-tail recursive version because of the overhead of processing the delays.

6.6 Combinators

The implementation described so far corresponds to a lazy SECD machine [Henderson 1980]. But it can also be used for a combinator machine [Turner 1979]. Thunks and code segments, being "first class" objects, can be passed as arguments and be returned as values. All the equality and call opcodes can take thunks as operands, evaluating them when needed as described above.

A thunk is evaluated only when required. When a thunk is evaluated, it may return a structure containing another thunk which itself requires evaluation when its value is needed.

The traditional combinators can be restated as predicates. Given the definitions:

$I = \lambda x.x$

$K = \lambda x\ \lambda y.x$

$S = \lambda f\ \lambda g\ \lambda x.(f\ x)(g\ x)$

we get the following predicate definitions (using *thunk/2* as described earlier):

```
comb_I(F,F).
comb_K(F,Z) :- thunk(comb_kk(F)/2,Z).
comb_S(F,Z) :- thunk(comb_ss(F)/3,Z).
```

with the auxiliary predicates:

```
comb_kk(X,Y,X).
comb_ss(F,G,Z)  :- thunk(comb_sss(F,G)/4,Z).
comb_sss(F,G,X) :- F(X,Z1), G(X,Z2), Z1(Z2,Z).
```

Here is a computation using *pluss* (1-ary addition) and *succ* (successor).

a = S plus succ 3

is compiled to the predicate definitions and calls

```
pluss(X,Z) :- thunk(plus(X)/2, Z).
succ(X,Z)  :- Z is X+1.
?- comb_s(pluss,A1), A1(succ,A2), A2(3,A).
```

resulting in:

A1 = *thunk comb_ss {parm0 = pluss}*

A2 = *thunk comb_sss {parm0 = pluss, parm1 = succ}*

A = *Z where* {`pluss(3,Z1), succ(3,Z2), Z1(Z2,Z)`}

leading to:

Z1 = *thunk plus {parm0 = 3}*

Z2 = 4

and finally (when evaluation of Z is forced all the way):

Z = A = 7

When code is written using the combinators *S*, *K* and *I*, the predicate calls to pred_S, pred_K and pred_I look just like normal predicate calls. When they are executed, their returned values are thunks which can be handled like any other objects. They will be evaluated only when needed and only as much as needed, possibly returning structures containing other thunks.

This definition of combinators requires that thunks be created only for the basic combinators *S*, *K* and *I* and of course for any other combinators (*B*, *C*, etc.) which are introduced to control the size of the compiled code (from which variables have been removed). All other definitions are done without reference to thunks.

The combinator machine uses a subset of the inference machine's instructions in a small number of ways, so some optimizations are possible. Call instructions invariably have two arguments (single argument and result), so the "load arguments, push new variable for result, call, pop result" sequence could be optimized into one instruction. By adding these optimizations to the machine, the programmer is given the flexibility of efficiently using either the lambda machine or the combinator machine models of functional computation – or mixing them – as the problem requires.

In Turner's combinator machine, combinator reduction occurs in-place. Turner observed that when an expression occurs within a function, it will only be evaluated once, the first time it is needed. Because logical variables share, we get the same effect with our predicate translations of combinators. In the combinator machine, values do not actually get replaced – rather, new values are computed and the old values are abandoned (and eventually garbage collected).

7.0 Design decisions

Quieta movere magna merces videbatur.

(Just to stir things up seemed a great reward in itself.)

— Sallust, *Catiline, 21*

Some design decisions are done for a good reason; others are simply arbitrary decisions and some are done just to simplify the implementation. Unfortunately, it is often not obvious to the reader which category a particular decision falls into.

The main design decision was to create an alternative abstract machine to the Warren Abstract Machine (WAM). Most implementations of Prolog use WAM with a few minor changes, typically adding some new instructions. However, I felt that the WAM was already too complicated and that it would be further complicated by adding delaying features.

7.1 Comparison with the Warren Abstract Machine instructions

In Prolog, here is a typical predicate which applies the predicate q to each element of a list:

```
p([], []).
p(Hd.Tl, HdX.TlX) :-
    q(Hd, HdX), p(Tl, TlX).
```

Here is xpPAM code:

```
  swXVNL  f0,n0,n2
  fail             % none of the 3 below
  goto    var      % parm0 is var
  goto    nil      % parm0=[]
                   % continue to lst case
lst:               % parm0=Hd.Tl
  eqlst   f1,n1,n3 % parm1=HdX.TlX
  link
  push    f2       % save Tl
  push    f3       % save TlX
  call    q/2      % q(Hd,HdX): regs 0 and
                   % 1 are already set
  pop     n1       % restore TlX
  pop     n0       % restore Tl
  unlink
  lCallSelf        % p(Tl,TlX)
nil:
  eq      f1,[]
  return
var:
  pushB   v0       % make the choice
  pushB   v1       % point by saving
  pushV   v2       % all active regs
  mkCh    else
  eq      f0,[]    % parm0=[]
  goto    nil
else:
  popB    n2       % restore the
  popB    n1       % active regs
  popB    n0       % on failure
  eqlst   f0,n0,n2 % parm0=Hd.Tl
  goto    lst
```

And the equivalent WAM code (courtesy of Saumya Debray):

```
  switch_term  var, nil, lst
var:
  try_me_else else
nil:
  get_nil     1
  proceed
else:
  trust_me_else_fail
lst:
  allocate          % create an environment.
  get_list    1     % arg 1, in reg 1, is a list.
  unify_tvar 1      % Hd is a temporary, put it in register 1; it'll
                    % be needed there for the call to q/2.
  unify_pvar 2      % T1 is a permanent, save it at
                    % displacement 2 in environment.
  get_list    2     % arg 2, in reg 2, is a list.
  unify_tvar 2      % same comment as for Hd.
  unify_pvar 3      % T1X is a permanent, save it
                    % at displacement 3 in environment.
  call        q/2   % notice args are in proper positions
  put_pval    2, 1  % move T1 into register 1
  put_pval    3, 2  % move T1X into register 2
  deallocate        % get rid of environment
  execute     p/2   % last call
```

Even though the WAM instructions are more complex than mine, more of them are required in the inner loop (10 vs. 13); for deterministic append, the difference is even more dramatic: xpPAM has just 3 instructions in the inner loop against 8 for WAM (these can be optimized to 2 and 7, respectively). Both have about the same number of memory and register references (in addition to call frame allocation and deallocation, xpPAM has 2 references to the heap and 4 to the execution stack; WAM has 9 memory references [Tick86]). It is difficult to draw any general conclusions from this example; WAM and xpPAM design appear to have similar efficiency.

7.2 Environments on the execution stack

Warren observed that passing arguments in registers rather than in stack frames has two advantages: tail recursion optimization can be easily performed and stack frames do not need to be created for unit clauses. However, his design keeps local variables in the stack. I have chosen to keep pointers to local variables in registers and to copy them to the execution stack only when they must be preserved across calls.

At first glance, my design appears less efficient. Although there are certainly cases where one or the other design is better, I believe that for "typical" programs [Matsumoto 1985], the two designs are similar in efficiency. In practice, only a few registers need to be saved around a call. In WAM code these registers would have to be loaded from the local or global stack anyway, so the number of executed instructions and stack references are about the same.

The push/pop around a call in my design does not only save values over predicate calls; it also puts values into the correct registers. This simplifies compiler design because register allocation need only consider where the registers are needed between two adjacent calls – the compiler (exclusive of code for handling delays and for detecting deterministic predicates) is about 600 lines of Prolog (which took a week to write and debug). The compiler seldom has to move the contents of one register to another; in WAM, instructions like *put_pval* appear quite often.

On a conventional machine, there is an interesting advantage to extending the heap or stack one cell at a time. If the next address can be made invalid, an addressing exception will happen when there is no more room on the heap or stack. However, an environment requires extending the stack a number of cells at a time and the environment cells must be contiguous, so every time an environment is built, there must be a test for stack overflow. Thus, xpPAM's allocating from a free list can be more efficient than allocating from a stack.

7.3 Global and local stacks

WAM has two execution stacks: "local" and "global." In the local stack, pointers are always from newer to older. Any pointers which cannot stick to this discipline are put onto the global stack (everything on the global stack is considered to be older than the local stack).

WAM's local stack is pushed and popped like a conventional execution stack (subject to frames being "protected" by backtrack information). The global stack is popped only on backtracking. However, the global stack can contain unused cells – for deterministic computations, the global stack must be garbage collected (some people use *repeat, fail* loops to get this effect – see section 8.7, "Cut" on page 173). That is, the WAM global stack is not really a stack – it is a global heap which can automatically remove some entries only on backtracking.

WAM's use of the global stack allows for very good speed figures for queries such as deterministic append (the "standard" LIPS figures use naïve reverse, whose inner loop is deterministic append). WAM does not need to garbage collect the resulting list because the stacks will be popped after the query. Because XpPAM allocates everything in a heap, it suffers a speed penalty for such queries. It is not clear how much better WAM is than XpPAM for more "typical" programs – if garbage collection is needed, both should be about the same. If there are delayed predicates, XpPAM is superior because there is no need to "globalize" variables.

XpPAM's design is intended for maximum speed for deterministic predicates. "Shallow backtracking" is a special case, which is handled by *if-then-elses.* Full backtracking is considered as an unusual situation – if some backtracking efficiency must be given up to gain deterministic efficiency, I consider that to be a good trade. I therefore see no advantage in WAM's ability to chop the global stack on backtracking.

Whenever two uninstantiated variables are unified, the newer points to the older. In WAM, age comparison is done by comparing stack addresses – the global stack is always older than the local stack. Therefore, there are never any pointers from the global stack to the local stack. Variables are recorded on the "trail" (backtrack stack) when they are older than the current choice point. The net result is similar to xpPAM's which keeps an "age" within each uninstantiated variable.

Compilers for WAM must do some analysis to determine which variables are local and which are global. There is no need for this with xpPAM. The local/global analysis also depends on the order of goals. In the presence of delays, this does not work, so à delay must "globalize" all the local variables, giving worse performance than xpPAM.

XpPAM's design allows treating predicates as first class objects. This is somewhat trickier in WAM because of the need to preserve and restore the state of the current stack frame ("environment"). Also, xpPAM can delay and resume a predicate at any instruction whereas WAM is more difficult to delay after an environment (stack frame) has been allocated.

XpPAM does not distinguish between "local" and "global" variables as in WAM. In WAM code, about half to three-quarters of all memory references are to the global stack (extrapolated from [Tick 1986] and [Matsumoto 1985]). Because xpPAM keeps pointers to local variables entirely in registers, it can get similar performance without the complication of stack shadow registers (a hardware implementation for xpPAM would probably cache the cells pointed by the registers, instead).

7.4 Saving environments (WAM)

The WAM creates an environment on the execution stack whenever a predicate has more than one goal. The environment is used to keep information across goal calls – it corresponds to the registers which are pushed and popped across a call for xpPAM. In addition, WAM instructions must be able to access either registers or offsets within the environment. This complicates WAM's unification instruction set, compared to xpPAM's instruction set which always works with registers.

There are two cases when a WAM environment must be saved: when delaying a predicate and when creating a "thunk." For a WAM-based design, all local variables would have to be "globalized" first, to guarantee that there are no dangling pointers.

With the xpPAM design, there are no environments and hence nothing special needs to be done when a predicate delays or when a "thunk" is created.

7.5 Allocating from a list or from a stack

Because all the xpPAM object cells are the same size, allocating and deallocating on a linked list is as efficient as allocating and deallocating on a stack. Linked lists are easier to use when storage must be reclaimed (garbage collected). Here is allocation code (in C) for the two methods:

Stack

```
if (stack > top)
    error();
new = *stack;
stack++;
```

Free list

```
if (list->tag != free)
    error();
new = next;
next = next->tl;
```

As mentioned earlier, the `(list->tag != free)`test can be removed if an addressing exception will occur when there is nothing left on the free list (that is, the last element of the free list points at an invalid address; when it is used, an addressing exception occurs). The test for stack overflow is trickier because an addressing exception will only occur when the stack element is first used, which may be somewhat later. However, hardware can do stack overflow checking in parallel, so the two methods may have the same speed with special hardware[18]

Lists have the advantage that additional memory can be allocated as needed. However, a segmented stack architecture is possible – instead of initially allocating one large stack, a smaller stack can be allocated and new stack segments can be added as needed (all segments are chained together). For the WAM, care must be taken because the "ages" of variables are compared by comparing their addresses.

7.6 Cut

The original WAM design did not handle *cut*. WAM implementers have handled this by adding new *cut* opcodes and doing some source level transformation. For example,

```
p(X) :- q(X), !, r(X).
```

is transformed to something like

```
p(X) :- cutPoint(Z), p2(X,Z).
p2(X,Z) :- q(X), cutTo(Z), r(X).
```

where the *cutPoint* and *cutTo* predicates translate directly to the new *cut* opcodes.

---

18 On a conventional machine, if a page can be marked as invalid (that is, it will cause an addressing exception when read of written), stack overflow can be detected with a single instruction which simply attempts to read from the top end of the frame (this only works if the frame is less than half a page in size, unless multiple pages are marked as invalid).

If xpPAM allowed *cut*, such a technique would also work. However, it is not necessary. "Shallow backtracking" situations allow *if-then-elses* which can be handled by the *switch*, *skip* and *goto* instructions. In some cases, choice points must be created. The mkChAt instruction records the cut point information. A later rmChAt or cutAt does a "soft" or "hard" cut, respectively. Such instructions could also implement "remote fail" or "deep bail out" situations – these are definitely non-logical but are not prevented by xpPAM.

7.7 Code indexing

In deterministic predicates, considerable speed-ups are possible if code indexing can be done. These are similar to *switch* instructions in C. The WAM has special instructions for this which work only on the first parameter.

XpPAM has generalized the WAM indexing instructions so that it can be used with any parameter. As shown by the example in section 5.5, "Optimized compiling of a predicate" on page 98, this allows taking advantage of all possible *if-then-else* situations, thereby minimizing the number of "create choice point" instructions.

The WAM *switch* instruction simply does a branch based on the object tag of the first parameter. Usually, the next instruction is a list element unify which must check the tag again. xpPAM combines these two functions so that its *switch* instruction is a generalization of the *unify list element* instruction. Similar generalizations are possible for structures and atoms.

7.8 RISCs, CISCs and in between

Because `xpPAM` uses only registers for operands, it is a little more RISC-ish than WAM which allows either registers or environment slots for operands. On the other hand, some of `xpPAM`'s instructions are more complex, such as `swXVNL` which combines two of WAM's instructions (*switch_on_term* and *unify_list*) into one. There is little point in making claims about degree of RISC-ness; what counts is performance and both designs seem to promise similar performance (of course, I think that my design is better).

Both WAM and `xpPAM` may benefit from "reduced instruction set" (RISC) design (see [Mills 1986] for an example). `XpPAM` has used this philosophy in a few places. Sometimes simpler instructions are better; in other places, complex instructions are better.

- The separate `link`, `push`, `call` and `pop` instructions are used to do calls; a more CISC-ish design (as in the VAX design for conventional computers) would combine these into one or two instructions (say, a *call* with a list of registers to be saved and restored[19]). The reasons for this decision are:
  - It is faster to use the separate instructions, even for an interpreter (the interpreter loop is faster than a *for*-loop for indexing over the registers to be saved).
  - It allows more flexibility in the compiler, so that a register can be `push`ed as soon as possible, to allow it to be re-used. In some cases, values can be saved on the backtrack stack instead.

---

19 The interpreter loop turned out to be faster than a *for*-loop over the operands!

- The `mkCh`; `pushB`; `pushB`; ... sequence is faster than a *make choice* instruction which encodes the registers to be pushed onto the stack. An additional advantage is in compiling *if-then-else* code (see section 5.6, "Shallow backtracking" on page 105) where the backtrack stack is used to save values across a call, saving one set of `pushes`.
- Earlier version of the implementation had separate *add reference*, *new free*, etc. instructions and no *v*, *n* or *f* annotation. This complicated the compiler and produced unification instructions that were less efficient. For example, the sequence *new 1; eq 1,2* was used to copy register 2 into register 1, requiring a general unification. To fix this, *copy* instructions were created – but they were just special cases of the `xpPAM` "`eq`" instruction.
- Earlier versions had a separate *setFail* instruction which put a failure address into a status register; whenever a unification instruction failed, the machine jumped to the failure address. The code at the failure address undid all the unifications and then executed a much simpler *fail* instruction. The advantage of a *setFail* instruction was that it simplified the *fail* instruction and it cut down on the number of opcodes (for example, `eqSkip` can be handled by a *setFail* plus `eq`). However, *setFail* was removed because it resulted in longer instruction sequences and because it greatly complicated the compiler.

A possible optimization to `xpPAM` is to add a variant of the *v* and *f* operand annotations to indicate that the register cannot point to a reference cell. This can speed up unification, although I do not know by how much. In WAM, such an optimization could be added by defining new instructions – the speed-up for WAM is likely to be better than that for `xpPAM` because it would speed up the *unify list element* instruction which immediately follows a *switch* instruction.

7.9 Instruction format

Both WAM and xpPAM are deliberately vague on the exact format of instructions. This is to be expected, because they are *abstract* machine designs. The instruction layout will depend on the underlying architecture of the implementation. As with most computer designs, there is some trade-off among simplicity, speed and compactness.

Both WAM and xpPAM have a fairly small number of instructions, supplemented by "foreign" instructions ("builtin" in xpPAM). This makes an interpreter fairly easy to write (a few thousand lines of code, plus memory management). XpPAM also has *n*, *v* and *f* annotation on its operands. Decoding these takes considerable time. Instead, the flags could be combined with the opcodes, resulting in perhaps 300 instructions (many flag combinations are impossible for most instructions). Of course, this makes the interpreter much larger but a doubling of execution speed is possible.

Some other tricks are possible for interpreted abstract machine code. Instead of storing register numbers, register offsets (into the register vector) can be used to save shifts; pointers to the registers can produce greater savings, at the cost of larger code size. Similarly, opcodes can be replaced by pointers to the actual interpreter code. As more of these techniques are applied, the result approaches threaded code. Unfortunately, threaded code interpreters are not very machine independent – if a portable implementation is desired, some significant performance degradation must be expected.

7.10 Functor and list storage

For simplicity, my implementation of xpPAM stores functors like lists. The only difference is a flag in the first element which distinguishes the two. The advantage of this technique is that constructs such as "`F(A)=g(x)`" are allowed (unification is the same as in "`[F,A]=[g,x]`"). The disadvantage is that more storage is used. If pointers are 4 bytes (12 byte object table entries) and that the machine requires 4-byte alignment of objects in *xArea* (to handle the back-pointers), the string "`f`" requires 20 bytes (12 byte object table entry, 4 byte back-pointer, 2 byte string (null-terminated as in C), 2 byte padding). Space for *nil* is not shown as it is pointed at by many list elements.

# args	Functor bytes	List bytes	Functor	List
0	40	32	`f()`	`[]`
1	64	64	`f(a)`	`[f,a]`
2	88	96	`f(a,b)`	`[f,a,b]`
3	112	132	`f(a,b,c)`	`[f,a,b,c]`

As can be seen, there seems to be little advantage of storing functors in *xArea* unless they have many arguments. Data in *xArea* must be compacted because they are of various sizes whereas data in *oTab* do not require compacting because they are all of the same size – the somewhat greater storage efficiency of storing functors as records is balanced by the greater increased time spent compacting *xArea* and by the greater complexity of unification.

Some additional thoughts on this are given in [Campbell and Hardy 1984]. I believe that **xpProlog** provides the best of both: notational flexibility and compactness plus simple implementation. However, there is nothing stopping an implementation of **xpPAM** from using the more traditional record-oriented method for functors. The next section discusses some other aspects of this issue.

7.11 Object cell size

My design is somewhat wasteful of memory because it allows only one object cell size. Some objects (such as nil and reference) have wasted space and other objects (such as strings and code segments) require an extra pointer into the extension area (*xArea*). Furthermore, I do not take advantage of putting objects within list cells and I do not provide compact list allocation such as *cdr*-coding.

The main reason for my scheme is to simplify implementation and to speed up unification. Suppose that a list cell could contain an uninstantiated variable object (instead of only a pointer to an uninstantiated variable object's cell). The action on unifying with a short integer would be to replace the object; unifying with a long integer or a string would require allocating a new cell and pointing at it. As most uninstantiated variables do eventually become instantiated, the only advantage of the more complex scheme is when dealing with lists which contain mainly short integers – but a lot of logic programming deals with strings which do not present any advantage in the more complex scheme. It should also be noted that the wasted space within an uninstantiated variable cell often gets fully used when the object becomes instantiated.

As the cost of increasing the complexity of unifying list elements more complex, *cdr*-coding could be used for storing list elements. Instead of allocating from a single free list, multiple free lists would be needed for the most common sizes; uncommon sizes would be kept in *xArea* (a similar technique could be used for strings). Functors would then appear like *cdr*-coded lists.

7.12 Reference counts and garbage collection

Garbage collection and reference counting are still somewhat contentious issues.

My implementation of **xpPAM** uses reference counting because reference counting frees unused memory as soon as possible. Reference counting also adds confidence to the correctness of the **xpPAM** implementation – errors in reference counts show up very quickly. As [Knuth 1973] notes:

> Some of the greatest mysteries in the history of computer program debugging have been caused by the fact that garbage collection suddenly took place at an unexpected time during the running of programs that had worked many times before.

A common implementation strategy for Prolog uses local and global stacks. **XpProlog** uses registers and the call save stack instead of the local stack and a heap instead of the global stack. Stack-like operations are as efficient on this heap as they would be on a stack. Using the heap allows freeing value cells which would become "trapped" within a stack [Bruynooghe 1982] and simplifies the implementation of delay which would otherwise require a branching stack.

Free annotation in xpPAM allows freeing objects as soon as they are no longer needed. These instructions are easily generated automatically at compile time. For example:

```
qsort(Unsorted, Sorted) :- qsortx(Unsorted, Sorted, []).
qsortx([], Sorted, Sorted).
qsortx(H.T, Sorted, Sofar) :-
    split(H, T, Lo, Hi),
    qsortx(Lo, Sorted, H.SortHi),
    qsortx(Hi, SortHi, Sofar).
```

In the last clause, *H* and *T* and the first parameter (*H.T*) can be freed after split; *Lo* and *H* can be freed after the first *qsortx*. *Qsort* is deterministic if the first parameter is fully instantiated and *split* is deterministic.

Reference counting also helps in implementing arrays. For example, the built-in predicate *chElem(A,I,V,A2)* takes an input array *A*, an index *I*, a new value *V* and replaces that element to produce the new array *A2*. If *A* has a reference count of 1, *chElem* can produce *A2* by modifying *A* in place rather than copying the entire array.

Reference counting and marking garbage collection seem to have about the same overhead [Krasner 1983] (about 15% if care is taken to avoid unneeded incrementing and decrementing the reference counts – this number has been verified for some simple examples with my implementation of xpPAM). Reference counting does not work well with circular lists but Prolog programs will not generate these, at least if they have unification with occurs check [Plaisted 1984]. A marking garbage collector could be used with xpPAM but care would have to be taken to ensure that the marking phase does not cause noticeable pauses – the pauses in **xpPAM** when the *xArea* is compacted are unnoticeable.

Considerable research is still being done on garbage collection. Two interesting possibilities are scavenging garbage collectors and incremental garbage collection. Both of these remove the pauses which garbage collections normally impose. Furthermore, they allow garbage collecting in parallel, if special purpose hardware is used. However, on conventional machines, these techniques do not seem to offer better performance than other storage reclamation techniques – parallel garbage collection would be disastrous on a time-shared machine.

Another possibility is to use some kind of hybrid scheme. A cell's reference count might be replaced with a flag which indicates that the cell is possibly referenced by more than one pointer (this is all that the optimized array predicate *chElem* requires). A general marking garbage collector would be needed to remove circular lists and to handle cells referenced by more than one pointer. However, such cells would tend to be more permanent, so the garbage collector would not be needed nearly so often and it would have less garbage to scan than if the simplified reference counting were not used. Combined with Bruynooghe's and Kluzniak's schemes for static analysis, this hybrid scheme may reduce heap management overhead to a very low number.

Extended pure Prolog

Conventional Prolog is inadequate for good logic programming. Some people have described it as the "Fortran of logic programming." When I think of Fortran, I am impressed by how advanced it was for its time and I am depressed because it has continued to be widely used, even though far superior languages have been invented later. Fortran was once a great advance in computer technology; it has become a hindrance.

It would be sad if Prolog were to become like Fortran. Until someone invents the Algol or Pascal of logic programming, we should consider some extensions to conventional Prolog which make it into a purely logical language:

- Adding flexibility to the execution order, including *if-then-else*, sound negation and coroutining (the delay notation for these is described in section 1.5, "Delay notation" on page 8).
- Extensions to Horn logic, including all solution predicates (*setof* and *bagof*).

- Logical arrays and logical input/output.
- No "impure," non-logical predicates.

The description is divided into sections:

- Section 8.0, "Execution order" on page 158 discusses how Prolog's top-down left-to-right execution strategy can be replaced by a more flexible strategy which **xpPAM** implements as efficiently as conventional Prolog's execution strategy.
- Section 9.0, "Coroutining, pseudo-parallelism and parallelism" on page 185 discusses how coroutining can be added to Prolog, including some examples which show the greater expressive power and efficiency that can be obtained. Extensions of coroutining to parallelism is also discussed.
- Section 10.0, "Extensions to Horn logic" on page 206 discusses negation, *if-then-else*, all-solution predicates, higher order predicates and meta-programming which contribute to making logic programming more expressive and practical. These extensions can be "purely" logical.
- Section 11.0, "Arrays and I/O done logically and efficiently" on page 222 discusses how arrays and I/O can be correctly incorporated into a logic programming language so that there is no need for predicates with side effects.

8.0 Execution order

Good order is the foundation of all good things.
— Edmund Burke, *Reflections on the Revolution in France*

8.1 Language issues

Conventional Prolog uses a strict top-down left-to-right execution strategy. A detailed description is given in [van Emden 1982]. This execution order has a number of undesirable affects:

- Some programs depend on the execution order to terminate successfully. These are inherently non-logical, typically using *cut* ("!") or *var*.
- Some programs terminate correctly, not depending on the execution order, but use non-logical I/O predicates which do depend on execution order.

Because of such programs, some optimizations cannot be applied. The situation is similar to conventional programming languages where an optimizer is prevented from transforming `f(x)+f(x)` to `2*f(x)` because of possible side effects inside `f(x)` (if you don't think this is a problem, just try this optimization on a program containing `rand()+rand()`).

XpProlog solves these problems by providing more expressive and safer control constructs than *cut* ("!") and *var*. XpProlog also provides logical I/O (see section 11.8, "Sequential Input/Output" on page 241).

Furthermore, more flexible execution order allows programming with coroutines. These are discussed in the next chapter.

8.2 Conventional Prolog's execution order

Conceptually, a Prolog program is a database of *clauses* which are processed in a rigid order. A clause is an expression of the form $H: - G_1,\ G_2,\ \ldots\ ,\ G_n$ where H is the *clause head* and G_1 through G_n make up the *clause body*, composed of *goals*. A *unit clause* or *fact* has no goal in its body (the ":-" is omitted); a *rule clause* has at least one goal in its body. Waterloo and subsequent IBM Prologs use " <-" instead of ":-" and "&"s instead of the commas separating goals.[20]

The clause's head and goals are all *atomic formulas* of the form $p(t_1, t_2, \ldots , t_n)$ where p is an n-place predicate symbol and t_1 through t_n are *terms*. Terms are of the form $f(t_1, t_2, \ldots , t_n)$ where f is an n-place functor symbol and t_1 through t_n are variables or terms. 0-place functors are *atomic*, that is names (strings) or numbers.

20 Both notations arose at about the same time, causing the original Marseilles notation (the head of a clause is marked with a "+" and the goals are marked with "−"s) to disappear. From a parsing point of view, the IBM notation is easier to handle, although it still overloads the dot (".").

A *query* is of the same form as a clause body. Conventional Prolog computes the solution to a query by trying the goals from left to right:

Try a goal: The database is searched from the beginning for a *matching* clause head:

- **Found:** The database position is remembered and variables are instantiated by unification (matching is often implemented by attempting a unification and undoing instantiations if the unification fails). If the clause body contains any goals, the goals are recursively tried from left to right. Once all the body's goals have been tried successfully, the goal has *succeeded.*
- **Not found:** The goal has *failed.* Computation *backtracks* to the previous goal which is re-tried.

Re-try a goal: Any variables which had become instantiated by unification are uninstantiated. The database is searched from the remembered position and execution continues as for "try a goal" (either "found" or "not found."

Trying a goal is very similar to calling a subroutine in a conventional programming language. Prolog can be efficiently implemented with a call stack which is slightly modified to allow for backtracking. As variables become instantiated, they are recorded so that they can be uninstantiated on backtracking.

This execution strategy can be improved in several ways:

- The clauses for a predicate can be grouped together so that only a small part of the database is searched when trying a goal. The clauses can be compiled to an abstract machine code (only for predicates whose clauses are not dynamically added to the database). Abstract machine code may be implemented directly in hardware, interpreted, or translated to machine instructions on conventional computers. Conventional wisdom states that interpreting abstract machine code is 10 − 20 times faster than processing the raw clauses. If the abstract machine code is transformed to machine instructions, a further 2 − 6 times speed-up is common. Hardware implementations allow cost reductions of one or two orders of magnitude.

- When a clause head matches and no other clause heads for the predicate can possibly match, some book-keeping information does not need to be kept ("choice points" for backtracking do not need to be created).
- When a variable which is newer than the last backtrack point becomes instantiated, it does not need to be recorded because it will automatically be deallocated when backtracking unwinds the stacks.
- When a predicate is called with all its parameters fully instantiated (no variables anywhere within its parameters), there is no need to ever re-try it – if it were to succeed again, it would provide no new information.
- When a predicate never instantiates anything within its parameters or does not instantiate a variable which is used by a later goal, there is no need to ever re-try it (this is a more general case of the previous case).
- When the last goal of a clause is tried, some book-keeping information is not needed on the call stack (*tail recursion optimization* (TRO)). If the goal is also in the last clause of a predicate, additional savings are possible.

These optimizations do not affect the order of execution; they simply speed execution and reduce stack usage. Details of some of them are given in section 5.0, "Compiling Prolog" on page 85.

Although easy to implement (at least, without the above optimizations), conventional Prolog's left-to-right execution strategy is inflexible.

- Unnecessary goals may be tried on backtracking. Sometimes this is merely very inefficient; in other cases, backtracking goes into an infinite loop.
- Excessive stack may be used. Tail recursion optimization can often prevent this.
- Negative goals can be executed correctly only in a restricted form which is not usually detected in conventional Prologs. Similarly, predicates such as *bagof* and *setof* must be restricted.
- Problems which can be naturally expressed as coroutines must be coded in an unnatural way. That is, `xpProlog` allows writing some programs in a natural way but conventional Prolog requires changing predicates into an unnatural form.

A subtle defect in conventional Prolog's execution strategy is that many programmers write predicates which depend on the execution order. Not only is this poor programming style, contrary to the spirit of logic programming, but it also inhibits some automatic program optimizations. Most predicates which do not depend on the execution order are penalized by the minority which do. A similar situation exists for conventional programming languages which allow side-effects inside functions which prevents optimizing `f(y)+f(y)` to `2*f(y)`.

8.3 Example of conventional execution order

The conventional execution order, described above, has the flavour of a conventional machine with backtracking thrown in. Execution proceeds sequentially, goal by goal. When a goal cannot be satisfied (it fails because no matching clause can be found), the machine is restored to an earlier state and an attempt is made to try an alternative clause. For example, suppose the following clauses exist:

```
a :- b1, b2, b3.
a :- b4.
b1 :- c1, c2.
b2 :- d1.
/* no clause for b3. */
b4. /* unit clause: no goals */
c1. /* unit clause: no goals */
c2. /* unit clause: no goals */
/* no clause for d1. */
```

We can transform this into a table:

Clause #	Head	Goal 1	Goal 2	Goal 3
1	a	b1	b2	b3
2	a	b4		
3	b1	c1	c2	
4	b2	d1		
5	b4			
6	c1			
7	c2			

We can treat the query "?-a" as clause 0 with the single goal "a."

1. Goal a has two alternatives: clause 1 (a:-b1,b2,b3) or clause 2 (a:-b4). The first alternative is taken.
2. Clause 1 (a:-b1,b2,b3) tries its first goal (b1).
3. Clause 3 (b1:-c1,c2) matches and in turn tries goal c1.
4. Clause 6 (c1) matches. It is a unit clause and succeeds. The next goal in clause 3 is c2
5. Clause 7 (c2) matches. It is a unit clause and succeeds. This causes goal b1 to succeed.

The next goal is b2 in clause 1. At this point the call stack is as follows (in a clause, the "▪" is after the goal being tried). Effects of tail recursion optimization (TRO) are not shown.

	Clause #	**Goal #**	**Alternative clause #**
`?- a. ▪`	0	1	-
`a :- b1, ▪ b2, b3.`	1	1	2
`b1 :- c1, c2. ▪`	3	2	-
`c1. ▪`	6	-	-
`a :- b1, b2, ▪ b3.`	1	2	-
`b2 :- d1. ▪`	4	1	-
`c2. ▪`	7	-	-

Nothing matches the goal d1, so it fails. There are no alternatives for c2 and b2, so they also fail. Clause 2 (a:-b4) is tried. This tries clause 4 (b4) and there is nothing else left to try, so the query succeeds:

	Clause #	Goal #	Alternative clause #
`?- a. ▪`	0	1	-
`a :- b4. ▪`	2	1	-
`b4. ▪`	4	-	-

As far as the correctness of the solution is concerned, the machine could have tried the clauses and goals in any order. The goals in clause 1 could have been tried in the order b3, b2, b1 (which would have failed more quickly) or clause 2 could have been tried before clause 1 (which would have succeeded without any backtracking).

8.4 A more flexible execution strategy

Instead of viewing the computation as using a stack, we could think of it as a series of goal re-writings, much like a Markov algorithm. The above computation could be given as the series of states (in each line, the "▪" is to the left of the goal which is about to be tried):

Apply clause	State
	`a ▪`
`a :- b1, b2, b3.`	`▪ b1, b2, b3`
`b1 :- c1, c2.`	`▪ c1, c2, b2, b3`
`c1.`	`c1, ▪ c2, b2, b3`
`c2.`	`c1, c2, ▪ b2, b3`

which gets to a failure point. Backtracking to clause 2 gives the alternative series:

Apply clause	State
	`a ▪`
`a :- b4.`	`b4 ▪`

This succeeds because all the goals are unit clauses (here there is only one such goal: `b4`).

In goal re-writing, any untried goal may be picked for re-writing. If it succeeds, it is replaced by all the goals of the matching clause – if there is none (a unit clause), the goal is removed from the set of goals which are candidates for re-writing. If it fails, the computation must backtrack to an earlier state and try different goals.

The easiest goal re-writing strategy to implement is one which proceeds strictly from left to right. The machine keeps a position marker within the re-write goals – everything to the left of the pointer has already succeeded by matching a unit clause and everything to right of the pointer has not yet been tried. The entire computation succeeds when there is nothing left to try. This is conventional Prolog's left-to-right, top-down strategy.

XpProlog uses an extended strategy which allows some goals to be *delayed.* Delayed goals are removed from the re-write goals until something causes them to be *resumed.* A resumed goal is put into the re-write goals immediately to the right of the pointer. In this way, every goal is tried, but possibly in a different order from what conventional Prolog would do.

In the above example, assume that goal c1 must delay until goal c2 instantiates something. Then the computation would be:

Apply clause	**State**
	▪ a ‖
a :- b1, b2, b3.	▪ b1, b2, b3 ‖
b1 :- c1, c2.	▪ c1, c2, b2, b3 ‖
delay c1	
c2.	{c1}, ▪ c2, b2, b3, ‖ c1
resume c1	c2, ▪ c1, b2, b3 ‖
c1.	c2, c1, ▪ b2, b3 ‖

The "‖" is used to mark goals which have been delayed and {...} marks where they were originally.

The delayed execution has now reached a failure point. Backtracking will cause execution to continue as for the non-delaying execution (above) because clause *c1* has no other alternatives.

In this example, the delay computation adds steps because goal c1 must first be delayed and then resumed. However, some problems can be sped up significantly by delays (see section 9.3, "Test and generate" on page 191).

With goal re-writing, backtracking always restores the machine to an earlier state. The pointer is moved left one goal and all the goals to the right of the pointer are thrown away. The machine then tries a different clause match for the next goal. If this fails, backtracking will throw away that goal and move the pointer left one more goal. The entire query will fail if the pointer is moved all the way to the left.

When backtracking occurs, delayed goals must also be removed. This strategy – a left-to-right with delays – will get to every successful state that a sequential left-to-right computation would produce, although possibly in different order.

If execution tries to go past the "‖," the answer is not "yes" or "no" but "maybe (insufficient information)."

8.5 Negation

Horn clauses contain only positive literals. Simple *not equals* can be considered as a convenient abbreviation for the infinite list of clauses:

```
a ≠ b.
b ≠ a.
a ≠ c.
1 ≠ 2.
[] ≠ [x].
   ...
```

This definition is sound, even if one or both of the arguments are uninstantiated. The goal X≠a will generate the solutions X=b, X=c, X=1, etc. on backtracking – a rather inefficient situation.

Many other useful predicates such as *less than* (<), call, name and *univ* (=..) are similar way to *not equals*. Such built-in predicates can also be thought of as containing an infinite number of rules like:

```
call(p(Q)) :- p(Q).
call(q(A,B,C)) :- q(A,B,C).
f(A,B) =.. [f, A, B]
g(A,B,C) =.. [g,A,B,C].
    ...
```

Conventional Prolog implements negation unsoundly; the *not* predicate ("\") is an attempt to implement negation as failure:

```
not(Test) :- call(Test), !, fail. /* fails if Test succeeds */
not(Test).                        /* succeeds if Test fails */
```

Negation as failure [Clark 1978] requires that the terms be fully ground (do not contain any uninstantiated variables). If *Test* is completely ground, the above definition of *not* is sound – it is equivalent to the definition given earlier with the "infinite" list of clauses. However, if *Test* contains any uninstantiated variables, this definition of *not* may produce a wrong answer. Consider the list membership predicate:

```
member(X, X._).
member(X, _.Rst) :- member(X, Rst).
```

The query

```
?- Z=4, not member(Z, [1,2,3]).
```

correctly succeeds (all the terms are fully ground) but the logically equivalent

```
?- not member(Z, [1,2,3]), Z=4.
```

fails because the call to *not* contains an uninstantiated variable. Here is the conventional, incorrect, execution sequence:

1. The `not` predicate tries `member(Z, [1,2,3])`. `Member`'s first clause succeeds, instantiating `Z=1`.

2. The *cut* ("!") in `not` removes all other choices for `member`.
3. `Z=4` is tried. Because `Z` is already instantiated to the value `1`, this fails.
4. Execution backtracks. The *cut* ("!") in `not` has already removed the other choices for `member`, so the entire query fails.

This can be remedied if *not* delays until its arguments are sufficiently instantiated. That is, *not* should act as if it had succeeded and be retried later when the variable which caused the delay becomes instantiated. Using the more flexible execution strategy given earlier, here is a correct execution sequence:

1. The `not` predicate delays trying `member(Z, [1,2,3])` because of the uninstantiated variable `Z`.
2. `Z=4` is tried, instantiating `Z` to the value `4`.
3. The `not` predicate is resumed by `Z` becoming instantiated. It now tries `member(Z, [1,2,3])`. Because `Z` already has the value `4`, this is equivalent to trying `member(4, [1,2,3])` which fails, causing `not` to succeed.
4. The entire query succeeds.

The above method of handling negation by delaying is not the only one possible. For example, [Voda 1986b] describes a more active approach in which environment can be propagated in a negative context.

8.6 Single solutions

For efficiency, the programmer may want *member* to succeed only once so that unnecessary backtracking is avoided – that is, the goal is "find the first *X* such that ...." Here is an attempted solution using *cut*. This technique is also often used to minimize stack usage. This poor programming practice is unnecessary if the Prolog implementation has tail recursion optimization (TRO). XpProlog has TRO.

```
member1(X, X._) :- !. /* succeed only with first match */
member1(X, Y.Rst) :- member1(X, Rst).
```

The query

`?- Z=2, member1(Z, [1,2,3]).`

correctly succeeds but the logically equivalent

`?- member1(Z, [1,2,3]), Z=2.`

incorrectly fails because the *cut* ("!") removes too many backtrack points. Member1 executes wrongly when Z is a variable. This can be made explicit, although rather clumsily, using *var*:

```
member1a(X, List) :- nonvar(X), !, member1b(X, List).
member1a(X, List) :-    var(X), !, member(X, List).
member1b(X, Y.Rst) :- nonvar(Y), X = Y, !.
member1b(X, Y.Rst) :-    var(Y), X = Y.
member1b(X, _.Rst) :- member1b(X, Rst).
```

But this is still not sufficient. `?- member([A,B], [[1,2], [3,4]]), A=3, B=4.` executes incorrectly. Instead, the *var_anywhere* predicate must replace the *var* to check for uninstantiated variables anywhere within the term, with increased computational cost.

A solution is to use an *if-then-else* which delays until the test is sufficiently instantiated:

```
member1c(X, Y.Rst) :-
    if X=Y then true
           else member1c(X, Rst) endif.
?- member1c(Z, [1,2,3]), Z=2
```

("true" is a predicate which always succeeds.)

The call to member1c provisionally succeeds with the test Z=1 delayed until Z becomes instantiated (Z=2). The resumed test (Z=1) fails and execution backtracks to the *else*. This tail recursively calls member1c which again provisionally succeeds with the test Z=2 delayed until Z becomes instantiated. The next goal Z=2 instantiates Z and the entire query succeeds. Backtracking will not re-do this computation because *then* removes the backtrack point for the *else* (this can be done safely because the test is fully instantiated and will therefore not produce any more information if it is retried).

If-then-else is a convenient abbreviation for a combination of *equals* and *not equals*:

```
member1d(X, Y.Rst) :- X = Y.
member1d(X, Y.Rst) :- X ≠ Y, member1d(X, Rst).
```

An optimizer can transform this into the *if-then-else* form. The optimizer can also notice that the equality/inequality test must not cause any variables to become instantiated: different unification instructions are used in this case. The computational cost of this *member1* predicate is linear with the size of the list, even with full checking for delay situations.

8.7 Cut

[O'Keefe 1985] and [Debray 1986] describe the uses of *cut*. The common uses are:

- Committing a clause.
- Forcing computations to be functions (that is, deterministic).
- Breaking out of failure-driven (*repeat/fail*) loops.

All of these can be eliminated from Prolog programs by *if-then-else*. An optimizing compiler can detect these situations in `xpProlog` programs and put cuts into the generated `xpPAM` code when they can be justified by maintaining the meaning of the original, purely logical, predicate.

Because `xpProlog` allows predicates to delay, the meaning of *cut* is very difficult to give. Consider

```
?- goal1, goal2, !, goal3.
```

where `goal1` has delayed and `goal2` has succeeded. The next goal is *cut* ("!") which removes all the alternatives within `goal1` and `goal2`. `Goal3` causes `goal1` to wake up by instantiating a variable that `goal1` is waiting on. If `goal1` then fails, everything will fail because there are no alternatives. However, an alternative in `goal3` might produce a different value which would cause `goal1` to succeed. The problem is that `goal1` did not really succeed – it only provisionally succeeded, pending more information (from `goal3` in this case). The implementation of *cut* must be very complex to handle such cases properly – they can be handled much more cleanly by *if-then-else*.

Tail recursion optimization removes the need for special *repeat* loops. In **xpProlog**, *repeat* is written

```
repeat(P) :- P, repeat(P).
```

which never consumes more stack than that required by *P* if *P* is deterministic. If it is not deterministic, the following is required:

```
repeat(P) :- if exists P suchthat P then repeat(P) endif.
```

where the *if-then-else* removes backtrack points. This is still logical but only the first solution for each invocation of *P* will be found.

Strictly speaking, *repeat* is not logical (if *P* has no side effects, it will always give the same result, so *repeat(P)* will either fail or loop forever. Typically, *repeat* is used to handle a read-execute-write loop. As **xpProlog** provides logical I/O with streams, *repeat* and *P* can be automatically transformed as described in section 11.8, "Sequential Input/Output" on page 241:

```
repeat(P, StdIn, StdOutRest, StdIn2, StdOut) :-
   P(StdIn, StdOut2, StdIn2, StdOut),
   repeat(P, StdIn, StdOutRest, StdIn2, StdOut2).
?- stream('<terminal>', input, In),     /* input stream from terminal */
   stream('<terminal>', output, Out), /* output stream to terminal */
   repeat(P, In, Out, In2, []).    /* In2 is remaining input */
```

where *StdIn* and *StdOut* are the input and output streams (the other *Std*___ variables are used to avoid calls to *append*).

Because many implementations lack tail recursion optimization and proper heap management, some programmers write the following for "infinite" loop programs (such as the top level of a read – eval interpreter):

```
repeat.
repeat :- repeat.
?- repeat,
   ...,
   fail.
```

Here, the *fail* chops the backtrack stack down and allows the execution stack to be freed. Because xpPAM fully supports tail recursion optimization and also garbage collects the global heap, there is no need to write in such a style.

The predicate

```
pred([], R) :- p1(R).
pred(a.Tl, R) :- p2(Tl, R).
pred(b.Tl, R) :- p3(Tl, R).
```

can be written using *if-then-else* as

```
pred(P1, R) :-
    if P1 = [] then p1(R)
    elseif P1 = Hd.Tl then
        if     Hd = a then p2(Tl, R)
        elseif Hd = b then p3(Tl, R)
        endif
    endif.
```

The programmer may use either form; the optimizer generates the same code for both forms.

If-then-else delays the predicate if there is a variable anywhere in the test, not just at the top level. This can be explicitly changed by adding "exists *variable-list* suchthat." For example:

```
lookup(Name, Value, FoundV, NameList, NewNameList) :-
    if exists FoundV suchthat member(Name-FoundV, NameList)
        then NewNameList = NameList
        else NewNameList = (Name-Value).NameList, FoundV = Value
    endif.
```

Here, *member* is used to try to find the Name-FoundV pair within NameList. If it is found, the name list remains the same; if it is not found, the new Name-Value pair is added to the name list. In both cases, FoundV gets the value associated with the name.

If-then-else can be used to make a general *first solution* predicate:

```
first(P) :- if exists P suchthat P then true else false endif.
```

There is no delaying within P. Once it succeeds, no other alternative is tried. *Member1* should be coded differently:

```
member1(X,List) :- if exists X suchthat member(X,List)
                then true else false endif.
```

which is less efficient than *member1c* because *List* must be scanned to ensure it has no uninstantiated variables.

8.8 Clause order

If a program uses only logical predicates, the order of clauses has no effect on the meaning of a predicate. However, the order can affect efficiency and, sometimes, whether or not it will terminate. Clause and goal order in xpProlog can be considered as strong hints to the compiler. XpProlog's optimizer is allowed to change the order of clauses in a predicate (an example is given in section 5.5, "Optimized compiling of a predicate" on page 98). Usually, the order will not be changed because the optimizer assumes that the most common cases have been put first.

If a predicate is deterministic, then only one clause will be tried; the order of clauses does not matter. If the clause is not deterministic, the clauses will be tried in the order that they are given, although the optimizer may be able to skip over some.

To allow compatibility with conventional Prolog, a predicate can be marked as requiring conventional ordered evaluation. For example this non-logical Prolog program

```
p(a,X) :- !, q(X).
p(b,X) :- !, r(X).
p(_,X) :- s(X). /* A ≠ a, A ≠ b */
```

is better written in **xpProlog** using the special *else* predicate as:

```
p(a,X) :- q(X).
p(b,X) :- r(X).
p(A,X) :- else(A), s(X). /* A ≠ a, A ≠ b */
```

which does not have any cuts. The **xpProlog** version does not depend on clause order for correct execution of the query `?-p(Z,1)` – on backtracking, it could generate Z=a, Z=b and Z=_[21] but the Prolog version would only generate Z=a.

21 Strictly speaking, the solution Z=_ should be the constraints $Z \neq a, Z \neq b$.

The programmer might prefer an *if-then-else* construct instead of the *else* predicate. The compiler would generate exactly the same code if the *p* predicate were written:

```
p(A,X) :- if A suchthat A=a then q(X)
      elseif A suchthat A=b then r(X)
                              else s(X)
      endif.
```

If the *suchthat*s were left out, the predicate would delay until *A* became instantiated – a similar effect could be produced by adding

```
?- proceed p(a?, x).
```

8.9 All solutions predicates

The problem of uninstantiated variables extends to second-order predicates such as *bagof* and *setof*. The meaning of conventional implementations of *bagof* and *setof* changes according to how the variables are instantiated when they are invoked, giving different meanings to programs according to the computation order (see section 10.3, "Setof, bagof" on page 211 for the definition).

The usual (unsound) implementation is something like:[22]

---

[22] This *bagof* (and **xpProlog**'s *bagof*) returns *nil* ([ ]) if nothing is found. Some people prefer predicates which fail if nothing is found. This can be easily defined:

```
bagof_fail(Proto, Pred, Result) :- bagof(Proto, Pred, Result),
                                   Result ≠ [ ]
```

```
bagof(Proto, Pred, List) :- (bagofAsserts(Proto, Pred) ; true), !,
                            (bagofRetracts(List) ; true).
bagofAsserts(Proto, Pred) :- Pred,
                             assert(bagof(Proto)),
                             fail. /* try another solution of Pred */

bagofRetracts(X.Rest) :- clause(bagof(X)), /* fails when no more */
                         retract(bagof(X)),
                         bagofRetracts(Rest).

setof(Proto, Pred, List) :- bagof(Proto, Pred, List1),
                            sortBag(List1, List).
```

The bagofAsserts first tries Pred and then asserts the prototype Proto. A fail causes backtracking to try the next possibility of Pred (backtracking over assert does not undo anything). The bagofRetracts simply picks up all the entries which have been added to the database, removing them as it goes. This definition will return [] if Pred fails.

There are several problems with this approach:

- It does not always have a declarative reading.
- The assert and retract predicates are quite expensive. This can be solved by using special purpose internal database predicates.
- The entire solution set must be generated, even if it is not needed. For example, if *not* is defined by bagof(_,Pred,[]), all the solutions to Pred are generated and then bagofRetracts fails as soon as the first element is found in the database. This also leaves un-retracted entries in the database, so *not* should actually be defined bagof(_,Pred,L),L≠[]. (*Setof* must still generate all the solutions because it must remove duplicates, usually by some kind of sorting.)

The sound implementation (using xpPAM) is very similar to that given above except that, when coded in abstract machine code, the solutions list can be generated directly. There is no need to enter and delete entries in a database. The xpPAM code is a little tricky and is not shown here. It is basically (see section 4.0, "Backtracking and delaying" on page 66 for a description of the instructions):

```
    delay as necessary
    make choice point (failOne)
    make fresh copy of Proto
    call Pred
    modify choice point (failTwo)
    append new list element with fresh copy of Proto
    fail /* doesn't remove copy of Proto on backtracking */
failOne:
    remove choice point
    unify [] with (tail of) List
    return
failTwo:
    fail /* the "add new list element" failed */
```

"Make fresh copy of Proto" means that a brand new copy is made, with no references to any free variables in Proto (see the *meta* and *unmeta* predicates in section 10.4, "Meta-variables" on page 212). As these variables are newer than the top choice point, they will left alone on backtracking; however, the "append new list element" must not be recorded on the reset stack so that it will not be undone when the fail causes backtracking to the choice point within Pred.

When there is no more choice in Pred, the choice point is used to go to label *failOne* which terminates the solution list with *nil.*

[Naish 1985c] advocates a notation which gives purely declarative readings to all solutions predicates. His design requires an implementation which can delay predicates until their arguments become sufficiently instantiated. XpProlog's meta-variables are a generalization of Naish's design. An all solutions predicate must delay until all non-local variables are ground. For example, in
`setof([X,Y], pred(Y,Z), Ans)`
the *setof* will delay until Z becomes ground. If this is written
`exists Z suchthat setof([X,Y] pred(Y,Z), Ans)`
there will be no delay. X and Y are "local" to the *setof* expression.

The problems with *not* and *member1* can be considered as special cases of the problems with *bagof* and *setof* because *not* and *member1* can be defined

```
not(Test) :- bagof(X, Test, []). /* nothing succeeded */
member1(X, List) :-
    bagof(X, member(X, List), X._). /* first solution only */
```

8.10 Non-strict execution order: delays

Most of the time, top-down left-to-right execution order is adequate for executing a Prolog program. But sometimes a more flexible execution order is needed.

Some conventional Prologs allow deviation from strict top-down left-to-right execution order by the *freeze* predicate. Using this, *member1* can be written:

```
memberle(X, Y._) :- freeze(X, freeze(Y, (X=Y, !))).
memberle(X, Y.Rst) :- memberle(X, Rst).
```

This only works if the scope of the *cut* is the entire `memberld` clause, not just the `freeze`.

Actually, two kinds of *freeze* are needed: one which delays until the argument is sufficiently instantiated (`freeze`) and one which delays until the argument is completely instantiated (`freeze_all`). Using this, we can produce one more definition:

```
memberlf(X, List) :- freeze_all(X, freeze_all(List, memberl(X, List)).
```

which is less efficient than `memberld` because it traverses the list an extra time, to detect uninstantiated variables. Also, `memberlf` may delay unnecessarily on a variable, for example, for `memberlf(1,[1,X])`. `Memberlc`, which uses *if-then-else*, is the most efficient and the easiest to understand.

Delays are the general mechanism for sound negation, single solutions, *if-then-else* and declarative all solutions predicates. Delays also allow the extensions to Prolog, including first-order quantifiers, described in [Lloyd and Topor 1984] and they allow coroutining with backtracking. The result is a very flexible design which permits a variety of programming styles. Delays are covered in section 9.2, "Coroutining example" on page 187. To take full advantage of the design, Prolog's syntax must be extended slightly – these extensions will be introduced as they are needed.

8.11 Input/Output

Input/output is a notoriously non-logical part of conventional Prolog. The most common design follows BCPL [Richards and Whitby-Stevens 1979]: input or output is always to the current *stream* and this stream can be changed at any time to be connected to a specified physical file. Predicates such as *display* and *read* manipulate the input/output streams using side-effects. Most Prolog texts give examples where the unwary programmer can be caught by I/O's illogical behaviour.

The main problems with conventional Prolog's I/O are

- Backtracking does not undo an I/O operation.
- If predicates delay, the order of I/O may not be what the programmer intended.
- The conventional I/O predicates work by side-effects, preventing some optimizations.

These problems can be solved by following a model similar to that in [Cheng 1986]. Input/output are still handled by streams but these streams are syntactically identical to lists. Meta-predicates associate these streams with physical files (section 11.8, "Sequential Input/Output" on page 241).

For debugging, it is often helpful to add output statements to predicates. XpProlog has retained this feature (which is non-logical) but its use is discouraged, except in debugging.

8.12 Efficiency

In my implementation of xpProlog (described by the xpPAM abstract machine), deterministic predicates are not slowed significantly by the delaying and backtracking features.[23] Depending on the problem, the machine is well suited to purely deterministic predicates and to complex generate-and-test predicates using coroutines and backtracking.

These implementation features are synergistic. For example, if determinism is recognized, then unnecessary choice points are not created. This speeds execution, allows tail recursion to take place, reduces stack and memory usage and decreases page faults.

[23] The current implementation does have an overhead of a few percent because each unification which instantiates something must check if a predicate has been delayed, depending on that variable. A better implementation would use a different tag for such variables and there would be no run-time overhead – they would just become another case in the jump-table used by the unification routine.

9.0 Coroutining, pseudo-parallelism and parallelism

Time present and time past
Are both perhaps present in time future,
And time future contained in time past.
— T. S. Eliot, *Four Quartets: Burnt Norton, 1*

Coroutining has been widely written about, but very few programming languages provide facilities for coroutining.

This chapter provides some examples of coroutining, to show how they lead to more natural programs than can be provided by purely procedural languages. In addition, coroutining follows naturally from logic programming's philosophy in that the order of execution does not affect the correctness of the solution. Thus, almost nothing must be added to the language to provide coroutining, although the implementation must change.

Coroutining can be considered as a special case of parallelism. Section 9.6, "Parallelism" on page 198 has a short discussion of the two paradigms.

Coroutining is not a new notion for logic programming. It is discussed in [Kowalski 1979]. Some other coroutining proposals are discussed at the end of this chapter.

9.1 Notation

Coroutining can be explicitly indicated by the "?" notation introduced in section 1.5, "Delay notation" on page 8 The "?" notation does not change the logical meaning of a programming except that the same program without "?"s may either fail to terminate or may run much slower.

Coroutining can be deterministic or non-deterministic. The deterministic case is similar to what is provided in conventional languages such as Simula-67. The non-deterministic case exists only for logic programming languages; its main use is in speeding up generate-and-test programs.

The "?" notation can be used within predicates or as separate *proceed* declarations (also described in section 1.5, "Delay notation" on page 8). *Proceed* declarations have the advantage that they can be written separately from a predicate's declarations.

[Naish 1985a] shows how similar declarations (*wait* declarations for MU-Prolog) can be generated automatically. The algorithm is surprisingly simple and can be implemented in a few hundred lines of Prolog. Of course, such an algorithm cannot generate delay declarations for all predicates (which would solve the halting problem) but it works for most cases encountered in practice.

9.2 Coroutining example

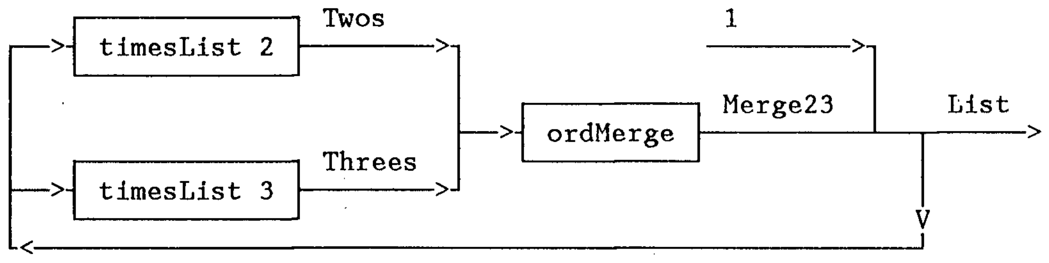
Here is a short, artificial example program to demonstrate coroutining. It is typical of a large class of programs which have a natural representation as coöperating parallel processes or coroutines. Such programs often are much harder to understand if they are transformed into a strictly sequential style. [Kowalski 1979 pp. 114-118] has other examples. [Dahl, Dijkstra and Hoare 1972] has more general comments on why coroutines are desirable, and many more examples.

The problem is to generate a list containing 1 and all numbers divisible only by 2 and 3, in ascending order with no duplicates: [1, 2, 3, 4, 6, 8, 9, 12, 16, 18, 24, ...]. This is similar to an exercise attributed to R. W. Hamming for which a sequential, less understandable, solution is given in [Dijkstra 1976]. [Henderson 1980 chapter 8] gives a functional lazy execution solution to this problem. I have restated his solution with predicates, giving an elegant solution using coroutines (degenerate parallelism).

The problem can be re-stated:

- 1 is in *List*.
- if N is in *List*, so are $2N$ and $3N$.
- no other values are in *List*.
- if *List* contains [..., A, B, ...] then $A < B$ (no duplicates, in ascending order).

Pictorially, the flow of data through the program is:



The predicate `timesList` generates a list of multiples in the third argument. For example:

```
timesList(2, [1,3,4], [2,6,8]).
```

The predicate `ordMerge` does an ordered merge of two ordered lists, removing duplicates. For example:

```
ordMerge([1,3,4], [2,4,5], [1,2,3,4,5]).
```

This program can be coded in `xpProlog`:

```
hamming(List) :- List = 1 . Merge23,
                 ordMerge(Twos, Threes, Merge23),
                 timesList(2, List, Twos),
                 timesList(3, List, Threes).
ordMerge([], M, M).
ordMerge(M, [], M).
ordMerge(A.X, A.Y, M)   :- ordMerge(X, A.Y, M). /* remove dup's */
ordMerge(A.X, B.Y, A.M) :- A < B, ordMerge(X, B.Y, M).
ordMerge(A.X, B.Y, B.M) :- A > B, ordMerge(A.X, Y, M).
timesList(_, [], []).
timesList(N, A.X, An.Xn) :- An is N*A, timesList(N, X, Xn).
```

This solves a set of simultaneous equations on *List* with *Twos*, *Threes* and *Merge23* as temporaries. Each of the *timesList* processes produces a list of 2 or 3 times its input list. The *ordMerge* process merges them (in order) and the resulting list is then fed back into the *timesList* processes. The duplicate removal is not completely general (it assumes that each of the input lists is in strictly ascending order) but is adequate for this problem. [Henderson 1980] outlines a proof that the above computation is well-defined.

Delays are necessary for *ordMerge* and *timesList*. *OrdMerge* must delay until its arguments are sufficiently instantiated for the ordering test (or until an argument is nil); *timesList* must delay until the arithmetic "`N*A`" is possible:

```
?- proceed ordMerge(a?.x, b?.y, m).
?- proceed timesList(n?, a?.x, list).
```

The program is very easily extended to deal with any number of `timesLists` by just adding more `ordMerge` stages – modifying the sequential solution is more complex.

Programmers who are used to sequential programs often have difficulty with the notion of predicates delaying and then restarting as their arguments become instantiated. Tracing the sequential execution of these predicates is very tricky. If these are thought of as parallel processes, some of the difficulty goes away – coroutines are just special cases of parallel processing where each process runs until it cannot proceed or until is provides a value which is needed by a suspended process.

With this in mind, here is how execution proceeds:

1. `ordMerge` delays on `Twos` and `Threes`.

2. the two `timesList` predicates produce their first elements: [2, ...] and [3, ...]. They then delay.
3. `ordMerge` wakes up and produces its first output: [2, ...]. It can proceed no further.
4. the two `timesList` predicates produce their next elements: [2, 4, ...] and [3, 6, ...]. They then delay.
5. `ordMerge` wakes up and produces its next output: [2, 3, ...]. It can proceed no further.
6. the two `timesList` predicates produce their next elements: [2, 4, 6, ...] and [3, 6, 9, ...]. They then delay.
7. `ordMerge` wakes up and produces its next output: [2, 3, 4, ...]. It can proceed no further.

And so on. In actual coroutining, the switching back and forth is done more frequently (as soon as a `timesList` predicate produces another element, `ordMerge` is resumed) but the net effect is the same.

The notion of delaying on variables is very similar to the concept of *monitors* [Hoare 1985]. Each variable which has caused a delay (by being marked using "?" or *proceed* notation) is like a monitor which suspends execution until sufficient information is available. The concept of critical region, however, is slightly different because there is no destructive assignment in logic programming.

These *proceed* declarations were generated automatically by using the method in [Naish 1984] from the original program. The *proceed* declarations are purely control information, separate from the logic of the program. However, they are necessary to ensure that the computation terminates.

9.3 Test and generate

Not only does coroutining allow for easier to understand programs, it can produce significant speed-ups. Generate and test programs typically are written

```
?- generate([a1, a2, a3, ... ], Solution), test(Solution).
```

where *generate* generates all potential solutions by permuting the problem's variables. The computational cost of this explodes factorially. Using coroutines, this can be written:

```
?- test(Solution), generate([a1, a2, a3, ... ], Solution).
```

The *test* predicate typically contains a series of inequalities and arithmetic predicates which have very large solution spaces. These are delayed. *Generate* then produces the first potential solution. It instantiates *Solution* sequentially from the first element. As the elements become available, the delayed tests are tried. If a test fails, it automatically removes all other potential solutions with that particular initial sequence – far fewer permutations are tried.

An example of this technique solves the arithmetic logic puzzle

```
  SEND            9567
+ MORE          + 1085
 ------  ====>  ------
 MONEY           10652
```

(S, E, N, D, M, O, R and Y are all different; M and S are not 0): Here is the program (adapted from [Colmerauer 1982]). The solution has no delay annotation because all the delaying is done by the "≠" predicates.

```
money(S, E, N, D, M, O, R, Y) :-
        different([S, E, N, D, M, O, R, Y]),
        M ≠ 0,  S ≠ 0,
        sum(C1, 0, 0,  0, M),
        sum(C2, S, M, C1, O),
        sum(C3, E, O, C2, N),
        sum(C4, N, R, C3, E),
        sum(0,  D, E, C4, Y).

/* sum with carry: C + X + Y = 10*C1 + Z */
sum(C, X, Y, C1, Z) :-
        goodCarry(C),  digit(X),  digit(Y),
        T is X + Y + C,
        split(T, C1, Z).

/* compute 2-digit number into its two digits: N = 10*C + X */
split(N, C, X) :- C is N / 10,
                  X is N mod 10.

/* enumerate the digits: */
digit(0).  digit(1).  digit(2).  digit(3).  digit(4).
digit(5).  digit(5).  digit(7).  digit(8).  digit(9).

/* For addition, the only possible carries are 0 and 1: */
goodCarry(0).  goodCarry(1).

/* Ensure that all elements in a list are different: */
different([]).
different(X.L) :- notMember(X, L),
                  different(L).

notMember(X, []).
notMember(X, A.L) :- X ≠ A,
                     notMember(X, L).
```

The *different* predicate sets up the delayed tests S≠E, S≠N, S≠D, ..., E≠N, ..., R≠Y. The *sum* predicate generates possible sums and carries for D+E=Y, N+R=E, etc. by backtracking through the possibilities produced by *digit* and *goodCarry*. As the values of S, E, ..., Y become instantiated, the inequalities are woken up. For example, the first *sum* generates an initial answer C1=0, M=0. This is immediately rejected by the constraint M≠0, so the other *sum* goals are never tried with those values for C1 and M. This drastically reduces the search space over what would be tried using conventional Prolog (by about 100 times).

The conventional Prolog solution would first have to re-arrange the goal so that the test predicates are after the generate predicates.

```
money(S, E, N, D, M, O, R, Y) :-
        sum(C1, 0, 0,  0, M),
        sum(C2, S, M, C1, O),
        sum(C3, E, O, C2, N),
        sum(C4, N, R, C3, E),
        sum(0,  D, E, C4, Y),
        M ≠ 0,  S ≠ 0,
        different([S, E, N, D, M, O, R, Y]).
```

All the possibilities produced by the *sum* predicates will be tried. Greater efficiency can be gained by interleaving parts of the different predicate with the *sum* predicates in the goal – but this obscures the program and is still not as efficient as the delaying solution.

Using such techniques can speed up programs by two or three orders of magnitude. I have observed programs dropping from an hour to a minute or even a second.

Even better speed-ups are possible by using constraints because integer arithmetic is complete (at least, Pressburger arithmetic is complete). Using constraints, the problem could be solved by:

```
money(S, E, N, D, M, O, R, Y) :-
        different([S, E, N, D, M, O, R, Y]),
        M ≠ 0,  S ≠ 0,
        0 is        (S*1000 + E*100 + N*10 + D
                  +  M*1000 + O*100 + R*10 + E)
          - (M*10000 + O*1000 + N*100 + E*10 + Y).
```

where *different* is as above. The xpPAM implementation cannot currently handle such constraints; it would need a generator for the values of *S*, *E*, ..., *Y*, such as the *digit* predicate given earlier.

9.4 Pseudo-parallelism

The structure of a typical parallel program is:

```
Requester          Server
                   initialize
send request       loop
                     wait for message
wait for reply       process message
                     send reply
process reply      endloop
```

This is a producer-consumer coroutine, with message passing instead of coroutine calls. No matter how much parallelism is available, the speed of the two processes is limited by the slower of the two. Using xpProlog, the coroutining process is:

```
coroutine(C, P) :- consume(L, C), produce(L, P).
produce(Hd.Tl, P) :- makeOneMsg(Hd, P),
                     produce(Tl, P).
produce([], P)    :- not exists Hd suchthat makeOneMsg(Hd, P).
                      /* end of produced list */
consume(Hd?.Tl, C) :- /* delay until list element and Hd instantiated */
     processOneMsg(Hd, C),
     consume(Tl, C).
consume([]?, C). % stop at end of list
```

The parameters *C* and *P* represent information which is used to control the consumption and production.

Consume immediately delays on *L*. *Produce* starts and continues with *makeOneMsg* which instantiates *Hd*, wakens *consume* and suspends *makeOneMsg*. *ProcessOneMsg* then executes. *Consume* repeats and delays on the uninstantiated tail of the list – this returns to the suspended *produce* and the cycle continues. The coroutine halts when *makeOneMsg* fails to produce a new element. Many kinds of control can be used; here, *C* and *P* are used to represent such control.

This program has one slight inefficiency. *Produce* will wake up *consume* as soon as it constructs a list element but *consume* will then immediately delay on the head. More efficient, but logically equivalent:

```
produce(List, P) :- makeOneMsg(Hd, P),
                    List = Hd.Tl,
                    produce(Tl, P).
```

The order of the *consume* and *produce* goals in *coroutine* is important for the interleaved execution; it is not important for the actual meaning of the program (we could have *produce* generate the entire list L first, then have *consume* process it – but this would use up enormous amounts of memory if the list were very large). The list L needs to exist just one element at a time because it is used solely as a communication channel. If reference counting is used, the list cells are freed as soon as they are accessed and so the entire list never exists, just the current element.

The *produce* and *consume* predicates are both deterministic and tail recursive, so the compiler can easily turn them into iterations. If *makeOneMsg* and *processOneMsg* are also deterministic, then the execution stack is bounded.

9.5 Correctness and completeness

The order of goal selection does not affect the correctness of the computation [Lloyd 1984]. However, it may affect the completeness. The programmer will want to be assured that any program which correctly terminates for conventional Prolog will also correctly terminate with **xpProlog**.

As long as there are no delays, **xpProlog** chooses goals in exactly the same way as conventional Prolog. For such programs, both languages have the same partial completeness.

Consider the simple query:

```
?- X < Y, X = 1, X = 2.
```

Depending on the Prolog implementation, this will either generate a run-time error (because "<" can only work with ground arguments) or it will fail (because "<" requires numbers as its arguments and uninstantiated variables are not numbers). Putting this query to `xpProlog` results in successful completion. Thus, `xpProlog` is "more complete" than conventional Prolog, if the delays are used properly.

Unfortunately, the programmer can take a working conventional Prolog program and add delay annotations so that it will not work in `xpProlog`:

- The delays may be too strict so that execution halts with some predicates still delayed. A trivial example, the query

  ```
  ?- X = 1.
  ```

  will succeed but the query

  ```
  ?- X? = 1.
  ```

  will delay indefinitely.

- The delays may cause an infinite loop. A trivial example is the query

  ```
  ?- X < Y, inf(A), X = 1, Y = 2.
  ```

  where *inf* is a predicate which does not terminate (for example, "`inf(A):-inf(A)`"). This query will fail, as described earlier, for conventional Prolog but will go into an infinite loop with `xpProlog` because the "`X<Y`" will delay, allowing the *inf* predicate go start execution.

To summarize, any pure Prolog program will produce the same results in `xpProlog`. Some pure Prolog programs will terminate correctly only in `xpProlog` – they will either produce wrong answers or will fail to terminate in conventional Prolog. However, the programmer must be careful in adding delays to programs – it is possible to cause an over-constrained program to deliver a "don't know" answer with a list of goals which cannot be computed:

- the goal(s) might succeed.
- the goal(s) might fail, allowing alternative goals to be tried (resulting in success, failure or a further "don't know").

9.6 Parallelism

Many problems which appear to be good candidates for parallelism and are written using parallel constructs are really disguised coroutines because critical sections must execute sequentially. A digital telephone switch has many thousands of processes (telephone calls) running concurrently. Most of these are suspended, waiting for an event such as someone picking up or putting down the telephone. The few active processes must all pass through one bottleneck: the circuit assignment process. Circuit assignment must be done sequentially and the capacity of the switch is determined by the speed of the circuit assignment algorithm.

This does not mean that some specialized problems cannot benefit from massive parallelism. Examples are equation solving by using relaxation techniques or searching large databases. These problems are characterized by needing only localized communication among processes:

Or-parallelism: the problem results in a large set of simple solutions which can all be tested independently. An example of this is a dictionary search for synonyms; another example is parsing with an ambiguous grammar. Generate and tests are not good candidates for or-parallelism because of their enormous search spaces; coroutining offers much greater speed-ups because it automatically prunes the search space.

And-parallelism: the problem can be broken down into two sub-problems which are completely independent. The overhead of detecting independent and-parallelism is about 10% [Hermenegildo and Nasr 1986], so parallelism results in significant speed-ups.

In fact, the telephone switch does gain some speed by using parallelism: peripheral processors can do some autonomous event processing. When the peripherals communicate the central call-processing CPU, they actually become part of a large coroutining system. For the central CPU, the advantages of parallel processing are not speed, but clarity.

Some parallel designs have retained standard Prolog with full backtracking [Hermenegildo and Nasr 1986]. Their intent is to retain the semantics of conventional Prolog, speeding execution when parallelism can be exploited. As **xpProlog** provides full pure Prolog with coroutining, it can easily be used in such a parallel machine – the coroutining predicates could transparently be executed on a fully parallel machine.

In contrast, guarded Horn clause languages (such as GHC [Ueda 1985], Concurrent Prolog [Shapiro 1983] and Parlog [Clark and Gregory 1984]) have abolished backtracking. Flat GHC has even discriminated against user predicates by not allowing them in guards. Standard Prolog can be implemented in such languages — but that can be said of even Fortran. Rather (attribution uncertain):

flat, safe, concurrent guarded Horn clauses = Occam + logical variable.

I believe that the indeterminacy of the guards — which has caused much semantic difficulty — is not a very valuable feature because the guards are usually mutually exclusive and can be transformed into an equally fast (or faster!) sequence of *if-then-elses*. The valuable feature of these languages is their ability to execute predicates in parallel, communicating via shared logical variables — a feature which can be exploited in any logic programming language which does not depend on execution order.

XpProlog's design for coroutining does not rule out parallelism. Because **xpProlog** allows only purely logical constructs, goals can be executed in any order and a coroutining program can be executed on parallel hardware with no changes. It may run faster — very often it will run slower because the cost of creating and destroying processes is greater than the saving producing by running in parallel.

9.7 Comparison with other designs for delaying.

MU-Prolog &refNaish 1985d] has a slightly less general form of delaying — it can delay only on entire arguments, not on the individual parts of them. In **xpProlog**'s notation, MU-Prolog can do `(Hd.Tl)?` but not `Hd?.Tl` (this is fixed in the successor, NU-Prolog). [Naish 1984] describes how to automatically generate *wait* declarations which are similar to but weaker than *proceed* declarations. His algorithm detects situations where backtracking would produce infinite solution spaces and then generates wait declarations to prevent them. The *hamming* program given earlier was originally tested in MU-Prolog; it will not run deterministically in MU-Prolog, even if some rather strange tricks are tried (supplied to me by Naish), because MU-Prolog has only the equivalent of simple top-level "?" annotations (because the *A* and *B* in *ordMerge* may be uninstantiated, the *less-than* predicate simply delays, allowing *ordMerge* to be called again – when *A* and *B* become instantiated, the *less-than* may fail, causing backtracking).

Prolog-II [Colmerauer 1982] uses a different method of delaying: the *freeze* (`geler`) predicate. This delays calling a goal until a particular variable has become instantiated. The test for delaying is thereby associated with the calling predicate instead of with the called goal. `XpProlog`'s method has the advantage of requiring the delay declarations in only one place; the extra flexibility of Prolog-II's method is not needed in practice. If desired, *freeze* can be defined in **`xpProlog`**:

`freeze(X?, Pred) :- Pred.` /* wait until `X` becomes instantiated, then call `Pred` */

Prolog-II can delay until one of several variables becomes instantiated ("or-delay") by a somewhat awkward – and not obvious – construct:

```
orFreeze(X, Y, Pred) :- freeze(X, Control = c),
                        freeze(Y, Control = c),
                        freeze(C, Pred).
```

The first *freeze* delays until X becomes instantiated; the second *freeze* delays until Y becomes instantiated. As soon as either of these variables becomes instantiated, the goal Control=c is executed which wakes up the *freeze* for Pred.

Unfortunately, this leaves unevaluated predicates lying around, so that we cannot distinguish between a program which is "hung" or permanently delayed and a program where some computational alternatives are not needed. XpProlog can do a cleaner *or-freeze*:

```
?- proceed orFreeze(x?, y, pred).
?- proceed orFreeze(x, y?, pred).
orFreeze(X, Y, Pred) :- Pred.
```

Freeze is very unwieldy for the *ordMerge* predicate given earlier. Using *freeze*, every call to *ordMerge(X,Y,M)* is replaced by

```
freeze(X, freeze(Y, ordM(X, Y, M))).
```

where *ordM* is defined by:

```
ordM([], Y, Y).
ordM(X, [], X).
ordM(X.A, Y.B, M) :- freeze(X, freeze(Y, ordMerge(X, Y, M))).
```

IC-Prolog [Clark, McCabe and Gregory 1982] has similar facilities as ours, adding unsynchronized parallel evaluation and parallelism with directed communication. Most practical parallel programs can be easily simulated with coroutines (see section 9.4, "Pseudo-parallelism" on page 194) so I do not consider my design to be deficient in this respect. There is no need for xpProlog to be implemented by coroutining – full parallel processing could be used.

IC-Prolog requires duplicating the bodies of predicates which can delay in more than one way like *append* or *ancestor*. It uses a variety of control annotations in the code (¬, //, !, ?, [,]) whereas **xpProlog** has a single separate control annotation (`proceed`). But these are not sufficient to handle the test-and-generate solution for eight queens (below). [Kluzniak 1981] has an interesting variant which seems to be more powerful than IC-Prolog's. He defines three control predicates:

spawn Call inserts a new process, in suspended state, immediate before the current process.

wait Variable delays until the variable becomes instantiated. The suspended process before the current process is activated.

yield suspends the current process and passes control to the next process.

Using this, the eight queens program is written (with "X?" being an abbreviation for `wait(X)`):

```
queens(X) :- spawn perm([1,2,3,4,5,6,7,8], X), safe(X?).

perm([], []).
perm(X.Y, U.V) :- del(U, X.Y, Z), yield, perm(Z, V).

del(A, A.L, L).
del(X, A.L, A.R) :- del(X, L, R).

safe([]?).
safe(X?.Y?) :- spawn nodiag(X, 1, Y), safe(Y).

nodiag(_, _, []?).
nodiag(B?, D?, (N.L)?) :- B =\= N-B, D =\= B-N, D1 is D+1, yield,
                          nodiag(B, D1, L).
```

Kluzniak's *wait* is the same as **xpProlog**'s ? The *spawn* is not needed in **xpProlog** because any predicate is potentially suspendable. **XpProlog** would write the *queens* predicate's goals in the order *safe, perm* (that is, the *safe* test is implicitly spawned because it immediately delays, waiting for the first element of the permutation to be generated). **XpProlog** does not require the *yield*s. As soon as a predicate instantiates a variable which has caused another predicate to delay (by a *wait*), the delayed predicate is activated. Kluzniak's design permits greater control but at the cost of requiring the programmer to know that the code will be used for coroutining (for example, *perm* must have a *yield* added to it for coroutining). Kluzniak also notes that coroutining between non-adjacent goals is difficult and he is unsure of appropriate backtracking behaviour (the details of **xpProlog**'s backtracking are in section 4.0, "Backtracking and delaying" on page 66).

The **xpProlog** solution to eight queens requires putting the *safe* goal before the *perm* goal, removing all the "?"s, *spawn*s and *yield*s and adding the following (which were generated automatically, using Naish's program):

```
?- proceed perm(?, -).        ?- proceed perm(-, ?).
?- proceed del(-, ?, -).      ?- proceed del(-, -, ?).
?- proceed safe(?).
?- proceed nodiag(-, -, ?).
```

These declarations make sure that the testing predicates (*safe* and *nodiag*) will not attempt to construct the solution list and the generating predicates (*perm* and *del*) will not go into an infinite loop on backtracking, generating longer and longer lists. The $X = \backslash = expr$ predicate is an abbreviation for *X2* is *expr, X* $\neq$ *X2* – it will delay until its arguments are instantiated. Incidentally, *nodiag* should not be "optimized" to delay until *N* becomes instantiated. With the *proceed* declaration above, *nodiag* will spawn a series of inequalities – the optimization would prevent these from being spawned. In predicates which produce something (such as *ordMerge*) should have the delays propagated to the head but those which test should not.

In summary, **xpProlog**'s "?" and *proceed* can do everything and more that the other delaying designs can do, but with simpler notation.

10.0 Extensions to Horn logic

... but exhaust the realm of the possible.
— Pinder, *Pythian Odes, III. 109*

Although Horn clauses are sufficient to handle any first order logic, they are not very convenient for some programming situations. In particular, the following have proven useful:

- Negation and *if-then-else.*
- All-solution predicates (*setof* and *bagof*).
- "Higher order" predicates.
- "Infinite" structures.
- Dynamic creation of predicates (meta-programming).

Negation and all-solution predicates are treated incorrectly by most Prolog implementations. .Sections 8.9, "All solutions predicates" on page 178 and 10.4, "Meta-variables" on page 212 describe language extensions for handling these properly.

Coroutining (discussed in section 9.0, "Coroutining, pseudo-parallelism and parallelism" on page 185) is not an extension to the logic but an extension to the control strategy. Arrays and logical I/O (discussed in section 11.0, "Arrays and I/O done logically and efficiently" on page 222) are *conservative* extensions to the logic in that they can be implemented using the existing mechanisms of the language. Mode and type declarations and equality theories (discussed in sections 5.10, "Speeding up deterministic predicates – modes and types" on page 112 and 6.5, "Equality: "is" and "="" on page 133) can also be treated as special predicates which are treated in a special way by the compiler.

10.1 Negation

Horn clause logic does not contain negation. But negation is very useful in practical programming. This section will discuss how negation can be handled properly in a logic programming language. Some of the material overlaps the section on execution order (see section 8.0, "Execution order" on page 158).

Consider the *member* predicate:

```
member(X, X._).
member(X, Y.Rest) :- member(X, Rest).
```

There are three ways that `not member(X,List)` can be processed:

1. Delay until everything in *X* and *List* are ground; call *member*; invert *member's* success or failure.
2. Execute *member*, in a special "negation" mode, with a list of variables which must not become instantiated (the predicate will delay rather than instantiate one of these variables).
3. Transform *member* to a new *notMember* and call that.

There is one disadvantage in the first approach: the query `not member(1,[0,1,X])` will delay unnecessarily. Furthermore, the delay is detected by recursively traversing all the arguments to *member* – this requires overhead similar to that required by the occurs check. The second approach adds significant complexity to the abstract machine. Transforming *member* gets around the problem.

Here is a transformation of *member* to its negative equivalent *notMember*. For any *X* and *List*, exactly one of *member* or *notMember* is true.

```
notMember(X, Tail) :- Tail ≠ (_._). /* or Tail = [] */
notMember(X, Y.Rest) :- X ≠ Y, notMember(X, Rest).
```

Here, the *not equals* ("≠") predicate does item by item delaying. When `notMember(1,[0,1,X])` is tried, the first clause fails (because the second argument is a list), so the second clause is tried. This succeeds, resulting in trying `notMember(1,[1,X])`. Again, the first clause fails. The second clause also fails, so the entire query fails, which is what we wanted. In contrast, using general *not* would have delayed until *X* became instantiated, resulting in unnecessary computation, or even never being tried if *X* never becomes instantiated.

This transformation can be done automatically. The method is:

1. Transform the predicate to an *if-and-only-if* form (using the *completion* [Clark 1978]).
2. Put in all universal and existential quantifiers.
3. Negate the predicate.

4. Work all the negations through to the innermost level so that *not* appears only in front of single predicates.
5. Transform the predicate back to Horn clause form.

Here is the transformation for *member*:

1. *member(X,List)* **iff**

   List = Y.Rest & (X = Y | member(X,List))

2. **forAll***(X,List): member(X,List)* **iff**

   exists*(Y,Rest): List = Y.Rest & (X = Y | member(X,List))*

3. **forAll***(X,List): notMember(X,List)* **iff**

   not **exists***(Y,Rest): List = Y.Rest & (X = Y | member(X,List))*

4. **forAll***(X,List): notMember(X,List)* **iff**

   forAll*(A,B):* $X \neq A.B$

   | **exists***(Y,Rest): List = Y.Rest &* $X \neq Y$ *& notMember(X,Rest)*

5.
```
notMember(X, List) :- List ≠ (_._).
notMember(X, Y.Rest) :- X ≠ Y, notMember(X, Rest).
```

In a typed system, a further transformation is possible because "`List≠_._`" is the same as "`List=[]`" However, there is a subtle difference between the two because the inequality will delay if it encounters an uninstantiated variable but the equality will instantiate a variable to *nil*.

Such transformations can be done at run time. XpPAM allows a "compile on first use" so that a predicate is stored in some external form until it is first tried. At that point, it is compiled (the external form is kept). Similarly, if the predicate is tried in a negative context, it can be transformed and then compiled.

Another situation where negation arises is when there is an either-or situation. For example, let us suppose that a fungus is always either a mushroom or a toadstool and never both. In sequent calculus (that is, Horn logic allowing multiple "goals" on the left hand side), this can be stated:

```
mushroom(X), toadstool(X) :- fungus(X).
```

From this we can derive two extended Horn predicates which state the situation, although without quite as much computational power because of the lack of general resolution:

```
mushroom(X)  :- fungus(X), not toadstool(X).
toadstool(X) :- fungus(X), not mushroom(X).
```

10.2 Closed predicates

The transformation of a negated predicate requires using the *closure* of the predicate [Clark 1978]. The assumption is that the programmer has written the program using *ifs* (":-" or "<-") but has really meant *if-and-only-ifs*. The *if-and-only-if* form is produced by simply or-ing the clauses for the predicate (step 1 in the previous section).

The negation transformation cannot be done in the presence of *assert* or *retract* on the predicate. To indicate this, xpProlog has a *closed* meta-predicate which states that the predicate will not be subject to any more changes (by *assert* or *retract*). An attempt to negate a non-closed predicate results in a run-time error.

Similar to closed predicates are *cannot fail* predicates. For such a predicate, *not* should always fail. When xpProlog compiles a cannot fail predicate, it adds one more "catch-all" clause which generates an error message. Many predicates have this cannot fail property – failure for these predicates indicates a programming mistake.

10.3 Setof, bagof

All solution predicates are often needed in practical logic programming. They are clearly second-order constructs outside of Horn clause logic and should not be simulated by using non-logical predicates such as *assert* and *retract*. All solutions predicates should be built into the logic programming language.

All solution predicates are general cases of negation and single solution predicates. For example, *not* could be defined as succeeding when the solution list is empty:

```
not(Test) :- bagof(X, Test, []). /* nothing succeeded */
```

Similarly, predicate *member1* which succeeds only once, for the first match can be defined:

```
member1(X, List) :-
    bagof(X, member(X, List), X._). /* first solution only */
```

Because all solutions predicates are second order, they require special second order variables. [Naish 1985c] describes a notation which gives purely declarative readings to all solutions predicates. The notation is similar to xpProlog's *exists* notation for *if-then-else*. He treats the problem as a control issue, indicating that execution should delays until certain variables become instantiated (or execution may proceed in spite of certain variables not being instantiated). XpProlog treats introduces second order variables to handle the situation. The next section discusses these *meta-variables*.

Section 8.9, "All solutions predicates" on page 178 describes a sound implementation of all solution predicates.

10.4 Meta-variables

Conventional implementations of *bagof* and *setof* misuse Prolog's logical variables. In the goal `bagof(Proto, Pred, Result)`, the variables in "`Proto`" (prototype) and "`Pred`" (predicate) are different from regular Prolog variables. They are really place holders: each time the predicate is evaluated, they are filled in with new logical variables. Such variables are common in higher-order predicates – I call them *meta-variables*. Meta-variables are limited in scope to the formulas in which they occur. They are place holders in formulas.

In the expression

```
setof(X, p(X), L1), setof(X, q(X), L2), ...
```

the two "X"s are distinct. The scope is obvious in all-solutions predicates, but there are other cases where the scope must be explicitly given.

Meta-variables also occur when predicates or formulas are manipulated. For example, a natural language query system takes sentence and transforms them into logical formulas (which look like predicates). At some later time, these formulas may be executed by filling in the variables. The input
All men like Mary.
might be transformed to the formula
```
all(X, man(X) -> likes(X, 'Mary'))
```
which can be tested by the query
```
?- if exists X suchthat man(X) & not likes(X, 'Mary') then fail endif.
```
by using the transformations in [Lloyd and Topor 1984].

When the logical form was produced, it got a new uninstantiated variable for "X." However, this was not quite what was wanted: a new meta-variable should have been used. XpProlog uses the *meta* predicate to create formulas containing meta-variables and *unmeta* to create predicates from formulas. For example, the following creates a logical formula in *Form*, turns it into a predicate and then executes it:

```
meta([X], all(X, man(X) -> likes(X, 'Mary')), all(MetaVar, Form)),
unmeta([MetaVar], Form, [Var], Pred),
if exists Var suchthat call(Pred) then fail endif.
```

Meta looks like a predicate but it must be handled specially by the xpProlog compiler. The arguments are:

1. the list of meta-variables.

2. the formula (using meta-variables).
3. the resulting formula.

Unmeta's arguments are:

1. the list of meta-variables.
2. the formula (containing meta-variables).
3. the resulting list of newly created variables.
4. the predicate with new logical variables substituted for the meta-variables in the formula.

Meta-variables are a generalization of Naish's notation for all solution predicates. They clear up ambiguities with logical forms and get rid of all need for *var* or "==" predicates.

Meta-variables must be marked. For example, it is reasonable to ask for the set of all [*A*,*X*] such that *pred(X)* – there is no need to have *A* instantiated before producing the set; *A* is not a meta-variable here but *X* is.

The notion of two kinds of variables (ordinary variables and meta-variables) opens up many issues in the design of **xpProlog** such as nice ways of marking them, scoping rules and automatically detecting them. These issues are beyond the scope of this report. I have taken the simple view that a variable is an ordinary variables unless:

- it is defined by a *meta* or *unmeta* predicate; or

- it appears in the prototype part of an all-solutions predicate and it has not been marked by an *exists*.

Thus, to get the set of all [*A*,*X*] such that *pred(X)*, one must write:

```
exists A suchthat setof([A,X], Pred(X)).
```

which the compiler transforms to:

```
meta([X], pred(X), MetaPred),
exists A suchthat setof([A,X], MetaPred).
```

To ease the compiler's job, it is illegal to use a meta-variable twice within a clause:

```
setof(X, P1(X), R1), setof(X, P2(X), R2).
```

would have to be written, substituting *Y* for the second *X*:

```
setof(X, P1(X), R1), setof(Y, P2(Y), R2).
```

10.5 Constraints vs. delays

Delays are passive. For example,

```
 integer(X), 3 < X, X < 5.
```

will simply delay until *X* becomes instantiated. However, these goals could be used to deduce that `X=4`. Alternatively, some delayed goals can be inconsistent. For example, $X > 3$ & $X < 5$ can never be satisfied. Yet xpPAM will simply delay in these situation, only able to test whatever value that *X* might be given.

Delayed predicates can be considered as *constraints* in that they constrain the range of values which an uninstantiated variable can be unified with. If we use set notation, the first example gives X being the set $\{Z \mid integer(Z)\ \&\ 3 < Z\ \&\ Z < 5\}$; the second example gives $\{Z \mid X > 3\ \&\ X < 5\}$. **XpPAM** keeps a list of all delayed predicates waiting for a variable in a list pointed from the variable – this can be considered as being very similar to the description of the set, but in a purely passive manner. That is, **xpProlog** merely tests for set membership instead of trying to built the set.

Constraints are easy to handle when finite sets are involved. When infinite sets are involved (such as the examples above), direct manipulation is impossible; the sets must be manipulated by their intensional meanings rather than by their extensional representations. This requires knowing additional theorems about the constraining predicates.

Multi-variable constraints are somewhat trickier to handle. For some predicates, these can still be processed. For example, $X^2 + Y^2 = 25\ \&\ integer(X)\ \&\ X > 0\ \&\ Y > 0$ has the two solutions $X = 3,\ Y = 4$ and $X = 4,\ Y = 3$.

Handling constraints is beyond the scope of this report. Constraints are very useful for solving certain kinds of puzzles which have arithmetic constraints (these puzzles may have practical uses, for example in helping compilers generate least cost code). It would be nice if a general mechanism could be found to allow specifying the theorems which pertain to constraining predicates. See [Jaffar and Lassez 1987] and [Voda 1986b] for further work on constraint programming.

Some problems are better solved by constraints and others are better solved by delays. Generally speaking, constraints are best for problems where a decision procedure can combine two or more constraints to produce a new constraint, such as solving inequalities (simplex programming, Diophantine equations, etc.). Delaying must be used for problems which cannot be handled this way. Also, delaying is useful for coroutining, which is a powerful program structuring tool but which can always be avoided by making more complex sequential programs.

10.6 The occurs check

Most Prologs do not implement the "occurs" check. The reason is simple efficiency: the occurs check requires a complete traversal of both structures being unified, raising the cost of unification from *O(N)* to *O(N^2)*. [Plaisted 1984] gives some methods of detecting when the occurs check can be safely avoided, by doing global analysis of the predicates.

The occurs check is necessary for unification to work correctly. The simplest situation is "`X=f(X)`" which has no possible solution (if *X* is required to be finite). For example, consider *append* for difference lists:

```
appendDiff(A-Z, Z-B, A-B).
```

The difference list can be turned into a regular list by a simple predicate, allowing *appendD* to be defined

```
diffToList(A-[] A). /* turn tail pointer into [] */
appendD(A,B,C) :- appendDiff(A, B, Z), diffToList(Z, C).
```

This produces the following results:

```
appendD(1.A-A, 2.3.B-B, C) ==> C = [1,2,3].
appendD(A-A, X, X)         ==> X = []
appendD(1.2.A-A, X, X)     ==> X = [1,2,1,2,1, ...]
```

Yet the last result is obviously wrong: it implies (by comparison with the second result) that `[]=1.2`.

Often, the occurs check is quite cheap because most unifications can be transformed into simple assignments, equality tests, splitting up list elements or creating new list elements. The occurs check is needed when general structures are unified, for example the difference lists above.

The cost of unification without the occurs check is $O(N)$ whereas with the occurs check it is $O(N^2)$. It should be noted that the occurs check makes the difference list *append* work in the same time as the ordinary *append.* Checking for delays in *not* or *setof* also multiplies the cost by $O(N)$ – it is desirable to eliminate all such costs. The occurs check can be left out because in practice very few predicates require it; the cost of delaying with negation can be made acceptable by the technique discussed earlier (section 10.1, "Negation" on page 207).

Prolog-II takes a different approach to the occurs check by allowing "infinite" structures called "rational tree":

```
X=f(X)   ===> X=f(f(...(...)...))
X=1.X    ===> X=[1,1,1,...]
```

This neatly removes the need for the occurs check, at the expense of changing the logic (the Herbrand universe is defined over *finite* structures). But there seems to be no reason to rule them out, especially as we are adding second order predicates (*setof* and *bagof*) and we can also handle potentially infinite list by the delaying mechanism. Indeed, infinite lists are useful for some kinds of grammar notation. However, they do complicate the implementation: output predicates must check for infinite structures and reference counting may not be able to collect all garbage.

10.7 Assert and retract

There are two main uses for *assert* and *retract:*

- To create new predicates "on the fly."
- To provide a database facility.

The first use is quite reasonable. For example, logic grammars (DCGs, etc.) are usually handled by being passed to a predicate which transforms the grammar notation to standard Prolog clauses, which are then added to the clause database. Such a use of *assert* is safe if it does not modify any predicate which is executing.

Assert can be used for meta-programming. As a simple example, logic grammars can be processed by a translator (a logic program, of course) to produce a new logic program. This is completely safe if the translator simply outputs the new logic program which is subsequently read in and executed, independent of the translator. But this is inconvenient, so `xpProlog` imposes no such restrictions; the user must be careful that an executing predicate is not modified.[24] And *retract* is provided so that a predicate can be removed before being replaced using *assert*s.

XpProlog implements *assert* by a "compile on first use" strategy. Whenever a clause is added to (or removed from) a predicate, the compiled code for the predicate is thrown away and replaced by code which invokes the compiler. The next time the predicate is called, the compiler is invoked – the resulting code is then kept and executed.

The second use of *assert* and *retract* is poor. For one, thing, the clause database is usually inefficient for general purpose database manipulations. Secondly, the predicates which do such operations rely on the side effects of the *assert* and *retract* "predicates" – that is, such predicates are not declarative.

A correct logical treatment of databases is briefly discussed in section 11.7, "Databases" on page 240 which gives a completely logical use of databases in which backtracking will remove any changes to the database . This does not mean that a database cannot change with time – after a query is completed, any changes to the databases (and other files) are made permanent.

XpPAM also provides a simple kind of associative table, defined by the predicates *tableAssign*, *tableFind* and *tableDelete* (the details are given in section Appendix D, "Built-in predicates" on page 300). These can have a declarative

[24] In fact, the abstract machine is likely to go wildly wrong if an executing predicate is modified because it assumes that a predicate will not change while it is executing.

reading such that changes are undone on backtracking. Associative tables are efficient internal databases.

11.0 Arrays and I/O done logically and efficiently

The simplest and most natural way to keep a linear list inside a computer is to put the list items in sequential locations, one node after the other. ... This technique for representing a linear list is so obvious and well-known that there seems to be no need to dwell on it at any length. ... It is important to understand the limitations as well as the power of the use of sequential allocation.
— [Knuth 1973]

This chapter can be read as a separate essay. It describes how arrays can be added to a logic programming language using only logical constructs and how this can be done efficiently. The discussion of arrays is as an example of how, by applying a little thought, a language designer can both please the purists and the pragmatists. The purists want constructs which are purely logical; the pragmatists want constructs which are expressive and efficient.

11.1 Introduction

Some logic programmers believe in using "pure" logic programming regardless of its efficiency; some others believe in efficiency regardless of its affect on the declarative reading of programs. Efficiency should not be an over-riding concern when writing programs, especially in a "very high level language"; nor can efficiency be completely ignored. XpProlog walks a middle path, allowing the programmer to pick appropriate data structures without having niggling worries about the efficiency of the resulting program. The idea is similar to that of the SETL project [Schwartz 1975] in which the programmer picks logical data structures which are natural for the problem and the compiler optimizes the data structures into particular implementations – the declarative reading of programs is retained while attaining acceptable efficiency.

Poorly thought out attempts to increase the efficiency of logic programs can actually backfire. Some programs can be sped up by a factor of 100 or more if a more flexible execution order is allowed (section 9.2, "Coroutining example" on page 187). Such optimizations can be used only if the programs are written in a purely logical (or declarative) style. When non-logical constructs are added to logic programs, these optimizations are not possible – the difficulties are similar to those encountered by optimizers for conventional programming languages when confronted with aliases caused by unrestrained use of pointers.

In logic programming, the problem specification is the program. Clarity of specification is important but efficiency must not be forgotten. Conventional programs often attain efficiency by using arrays and destructive assignments. These facilities are not generally available in logic programming languages –

destructive assignments are impossible within the declarative style of logic programming.

Some Prolog programs emulate arrays using lists or trees (see, for example, [Kluzniak and Szpakowiez 1985, pp. 113-120]). At best, these provide $O(\log N)$ access to arbitrary elements (N is the number of elements in the simulated array) whereas true arrays provide constant time ($O(1)$) access.[25] However, arrays cannot always emulate lists. Arrays always have a known length but lists may be of indeterminate length if the tail of the last element is not instantiated.

I will present some array manipulation predicates which have obvious declarative meaning. These predicates can be implemented efficiently. Automatic transformations can take advantage of situations where destructive assignment can be used.

In most current logic programming languages, I/O is implemented by highly non-logical predicates. Many people have suggested that *cut* can be eliminated from logic programming languages. If *var* and "==" are also eliminated (by some form of delay mechanism or by meta-variables), the only remaining non-logical predicates are those which do I/O. The logical treatment of arrays can be extended to files and databases, removing the need for any non-logical predicates. The only difference between arrays and files is where the data are

[25] This approach may not be entirely invalid: [Wise 1987] provides an argument that for certain numerical problems, trees are adequate. However, arrays have uses beyond matrix algebra and for these, constant time access appears to be crucial.

stored. Techniques which give efficient implementation of arrays also give efficient implementation of I/O.

With a rich choice of data structures and a syntax which supports them, programs can be written clearly, concisely and logically.

11.2 Array Operations

I will show the operations only for 1-dimensional arrays (vectors); extension to multidimensional arrays are "left as an exercise for the reader."

The elements of an array do not all need to be the same and uninstantiated variables are allowed inside arrays. Space and time efficiency can be improved if the compiler knows that all the elements will be the same, without any uninstantiated variables.

One particularly useful form of array is a *string*. A string is just an array of single characters. The operations which can be applied to a string are the same as those for an array. The typing predicate `string(S)` has no effect on the logical reading of a program; it merely increases efficiency by guaranteeing that all elements of the array are single characters (see section 5.10, "Speeding up deterministic predicates – modes and types" on page 112). Putting an uninstantiated variable into a string causes a delay and putting in anything else causes a run-time error.

First, the basic operations:

bounds(A, L, H) The array typing predicate. **L** is the low bound of **A** and **H** is the high bound. This predicate can either get the bounds of an array or create an array of the given size (with all uninstantiated elements). The high bound of an empty array is one less than the low bound – this follows from the definition of *length* (predicate `len` below) which is *high-bound–low-bound*+1.

elem(A, I, X) Extract one element: **X** is the **I**th element of **A**. That is, **X** is **A[I]**.

chElem(A, I, X, A2) Change one element. **A2** is the same as **A** except that the **I**th element has been set to**X**, as if `A2 := A, A2[I] := X`.

These are sufficient to define all other array predicates.

For notational compactness, expressions may exist inside predicate calls. An expression is enclosed in { ... } which is read "evaluate."[26] The expression is evaluated by using the "`is`" predicate ("`:=`" in Waterloo or IBM Prolog). Thus

```
factorial(N, {N * F}) :- factorial({N - 1}, F).
```

has the same meaning as

```
factorial(N, NF) :- N2 is N - 1,
                    factorial(N2, F),
                    NF is N * F.
```

Some array operators can be used in "`is`" expressions:[27]

---

26 There is no ambiguity with grammar notation's use of curly brackets – expressions can occur only within goals whereas grammar rule curly brackets can occur only around goals.

27 These are just extensions to the "built-in" arithmetic operations which could be defined

```
L is lob A :- bounds(A, L, _). /* low bound */
H is hib A :- bounds(A, _, H). /* high bound */
0 is len [].
L is len A :- len(A, L). /* "len" defined later */
X is A[I]  :- elem(A, I, X).
X is first A :- elem(A, {lob A}, X).
X is last A  :- elem(A, {hib A}, X).
A2 is A rangeFrom I :- rangeFrom(A, I, A2). /* "rangeFrom": below */
A2 is A rangeTo I   :- rangeTo(A, I, A2).   /* "rangeTo": below */
A2 is rangeFromSecond A :- rangeFrom(A, {lob A + 1}, A2).
```

Following are some standard array predicates. These are built-in for efficiency, but they can all be implemented with just `bounds`, `elem` and `chElem` Many other array predicates are possible. I have restricted myself to a list similar to those in [Dijkstra 1976].

swapElem(A, I, J, A2) Swap two elements. **A2** is the same as **A** except that the **I**th and **J**th elements have been swapped:

```
A2 := A, A2[I] := A[J], A2[J] := A[I]
```

or, using `chElem`:

```
chElem(A, I, {A[J]}, A1), chElem(A1, J, {A[I]}, A2)
```

```
X is X? :- atomic(X).
X is A+B :- A2 is A, B2 is B, arith(+, A2?, B2?, X2), X=X2.
X is -A  :- A2 is A,          arith(-, A2?,      X2), X=X2.
```

etc. where *arith* is a low-level predicate which expects all its arguments except the last to be instantiated. Note the usage of "?"s within *arith* which delay the entire clause until everything is sufficiently instantiated.

concat(A1, A2, A) Concatenate two arrays. **A** is the concatenation of. **A1** and **A2**. After this operation:

`bounds(A, {lob(A1)}, {hib(A1) len(A2)}).`

`Concat` can be used with the first two parameters uninstantiated, in which case it generates sub-arrays (or sub-strings) by backtracking.

subrange(A, L, H, A2) Extract a subrange of an array. **A2** contains the subrange of **A** delimited by **L** and **H**. If **H** is less than **L**, then **A2** is unified with an empty array.

loEx(X, A, A2) Low extend an array. **A2** is the same as **A** except that the low bound has been decreased by one and the new element is **X**. After this operation:

```
{lob A2} = {lob A - 1},
{hib A2} = {hib A},
subrange(A2, {lob A}, {hib A}, A),
{A2[lob A2]} = X
```

hiEx(A, X, A2) High extend an array. **A2** is the same as **A** except that the high bound has been increased by one and the new element is **X**. After this operation:

```
{lob A2} = {lob A},
{hib A2} = {hib A + 1},
subrange(A2, {lob A}, {hib A}, A),
{A2[hib A2]} = X
```

reshape(A, L, H, A2) Reshape an array with new bounds. **A2** contains the same elements as **A** but with different bounds defined by **L** and **H**. An error occurs (and the predicate fails) if **A** and **A2** have different numbers of elements.

For convenience, the following are defined:

low bound `lob(A, L) :- bounds(A, L, _).`

high bound `hib(A, H) :- bounds(A, _, H).`

length `len(A, {hib A - lob A + 1}).`

range from `rangeFrom(A, L, A2) :- subrange(A, L, {hib A}, A2).`

range to `rangeTo(A, H, A2) :- subrange(A, {lob A}, A2).`

low remove one element `loRem(A, {A[lob A]}, {A rangefrom lob A + 1}).`

high remove one element `hiRem(A, {A[hib A]}, {A rangeTo hib A - 1}).`

shift `shift(A, N, A2) :- reshape(A, {lob A + N}, {hib A + N}, A2).`

An array predicate fails if an index falls outside the range of the bounds. A run-time error may be preferable to simply failing because an out of bounds condition usually suggests an error in program logic.

Array constants are defined by specifying either the lower or the upper bound, plus a list of all the elements (because they are declarative, these predicates also convert arrays to lists):

```
arrayFrom(A, L, [X1, X2, ..., XN]) /* array "A" with low bound "L"  */
arrayTo  (A, H, [X1, X2, ..., XN]) /* array "A" with high bound "H" */
```

These predicates can be thought of as extensions of the standard “`name`” and “`=..`” (*univ*) predicates.

The empty array is denoted `[]`.

11.3 Transformation to allow destructive assignment

As a simple example, consider a predicate which increments each element in a list:[28]

```
incrList([], []).
incrList(A.Rst, A2.Rst2) :- A2 is A + 1,
                            incrList(Rst, Rst2).
```

For arrays, this is:

```
incrArray(A, A2) :- bounds(A2, {lob A}, {hib A}),
                    /* A2 has same shape as A */
                    incrArray2(A, {lob A}, A2).
incrArray2(A, I, A2) :- I > {hib A}.
incrArray2(A, I, A2) :- I =< {hib A},
    elem(A2, I, {A[I] + 1}), /* A2[I] = A[I]+1 */
    incrArray2(A, {I+1}, A2).
```

A general *incr* predicate is transformed to something like:

```
incr(Input, Output) :-
   if ground_array(Input) then incrArray(Pred, Input, Output)
                          else incrList(Pred, Input, Output).
```

---

28 This is a special case of the following Prolog idiom (which is much like the *mapcar* function in LISP):

```
map(Pred, [], []).
map(Pred, X.Rest, Y.Rest2) :- Pred(X, Y),
                              map(Pred, Rest, Rest2).
```

If *Pred* is deterministic when its first argument is ground, the transformations in this section can be applied.

The compiler can now generate the following Pascal-like code for the case when *A* is a ground array, by destructively modifying *A*:

```
procedure incrArray(A : array, var A2 : array) begin
    int I;
    for I := lob A to hib A do
        A[I] := A[I] + 1;
    A2 := ptr A; /* A2 points to (modified) A */
end procedure
```

Destructive assignment can only be used if *A* is not needed after *incrArray* is called. Destructive assignment could not be used in the goal

```
?- ..., incr(A, A2), p(A, A2).
```

because *A* is used by *p*. Instead, this must be transformed to (*copy* is defined later):

```
?- ..., copy(A, Ax), incr(Ax, A2), p(A, A2).
```

Destructive assignment is slightly more complex than in Pascal because of the possibility of backtracking. The Prolog abstract machine can record information for undoing the instantiation on the backtrack stack (called "trail" in the Warren Abstract Machine [Warren 1983]). If the value being instantiated (or, in this case, modified) is newer than the latest choice point, then nothing need be saved. If backtracking information must be recorded, the entire array *A* is not saved – only a pointer to *A2* and the changed value. Backtracking can then restore the single changed value.

11.4 Efficient implementation of array operations

There is an extensive literature on methods for storing data in arrays (see, for example, [Knuth 1973]). Although any of these storage methods can be used, I will assume that an array is kept as a triple, containing the low bound, high bound and a pointer to the data. The data are kept in a contiguous vector. The elements are all pointers to other objects (including uninstantiated variables). Thus, any element can be reached by an ordinary array indexing operation in constant time.

The following predicates access elements of arrays and are very cheap:

`bounds(A, L, H)` (also `lob(A, L)`, `hib(A, H)` and `len(A, L)`)
`elem(A, I, X)`

The following *subrange* predicates produce a new array which is some part of another array. They are also very cheap because they merely require creating a new object with different bounds, pointing to somewhere within the original array.

`subrange(A, L, H, A2)` (also `rangeFrom(A, L, A2)` and `rangeTo(A, H, A2)`)
`loRem(A, X, A2)`
`hiRem(A, X, A2)`
`reshape(A, L, H, A2)` (also `shift(A, N, A2)`)

The following require changing array elements to create a new array:

`chElem(A, I, X, A2)`
`swapElem(A, I, J, A2)`

```
concat(A1, A2, A)
loEx(X, A, A2)
hiEx(A, X, A2)
```

These operations are cheap if they are done in-place (destructively) but can be expensive if they require array copying. `LoEx` usually requires an array copy but it is not a common operation because arrays are usually built using `hiEx` or `concat`. `HiEx` and `concat` can be efficiently implemented (as in the XPL language [McKeeman, Horning and Wortman 1970]) by simply adding the new element(s) to the array if the heap space following the array has not been allocated. This is frequently the case because usually only one array is built up at a time. Thus, using `hiEx` to build an array is as efficient as pre-allocating the array (using `bounds`) and filling in the elements (using `elem`).

Array copying can be implemented:

```
copy(A, A2) :- bounds(A2, {lob A}, {hib A}),
               copy2(A, {lob A}, A2).
copy2(A, I, A2) :- I >  {hib A}.
copy2(A, I, A2) :- I =< {hib A},
                   elem(A2, I, {A[I]}),
                   copy2(A, {I+1}, A2).
```

This is nothing more than a statement of what it means for two arrays to be equal; that is, `A = A2` if the bounds are identical and the corresponding elements are identical.

Array copying should be avoided. But destructive assignment cannot be used everywhere because it can make programs non-declarative. For example, in Pascal:

```
A[1] := 'a';  A[1] := 'b';
```

is not the same as

```
A[1] := 'b';  A[1] := 'a';
```

as might be expected by a naïve translation to Prolog (the "," ("and") operator in pure Prolog is semantically commutative). However, `chElem` is declarative:.

```
chElem(A, 1, a, A2), chElem(A2, 1, b, A3)
```

is clearly the same as

```
chElem(A2, 1, b, A3), chElem(A, 1, a, A2)
```

if the Prolog implementation allows delaying goals (`chElem(A2,1,b,A3)` must be initially delayed because `A2` is as not yet instantiated).

The goals

```
chElem(A, 1, a, A2), p(A2)
```

can be replaced by destructive assignment

```
A[1] := a, p(A)
```

because the original array `A` is not used after it is modified.[29] If the original array were needed later, a copy would be needed:

```
copy(A, A2), A2[1] := a, ..., p(A2), ..., q(A).
```

which is clearly equivalent to

```
chElem(A, 1, a, A2), ..., p(A2), ..., q(A).
```

It is also logically correct because `copy(A, A2)` has the same meaning as `A = A2`.

Destructive assignment does not affect backtracking. In

```
..., p(A), chElem(A, 1, a, A2), q(A2), ...
```

29 More strictly, nothing else may point at A For example, if this were preceded by B=A and B were used later, then A could not be modified destructively.

if q fails and p has alternative clauses, p will be retried with the value of A before the destructive assignment. This is because destructive assignments to A (which create A2) are recorded on the backtrack stack and undone on backtracking.

In theory, it is possible to do a global analysis of all the predicates and determine which ones may use destructive assignments. Copy goals can then be inserted ahead of calls to these destructive predicates. Some of these copys may be unnecessary because a destructive predicate does not guarantee a destructive assignment, only the possibility. Moreover, the analysis is not simple because of subrange arrays. Some of the techniques of alias detection for conventional programming languages may be applicable. See [Bruynooghe 1986] and [Kluzniak 1987] for some preliminary work in this direction.

Another method of avoiding array copying is to associate a reference count with each array and make a copy only if the reference count is greater than one. The reference count must also include the counts of all subrange arrays. This technique ensures that array copying is done only if necessary. Some studies in [Krasner 1983] suggest that optimized reference counting has about the same overall cost as marking garbage collectors – by eliminating unnecessary array copying, reference counting may actually be more efficient than marking garbage collecting.

11.5 An example: Quick-sort

One of the favourite examples of logic programming is *quick-sort* for sorting a list:

```
qsort(Unsorted, Sorted) :- qsortx(Unsorted, Sorted, []).

qsortx([], Sorted, Sorted).
qsortx(Pivot.Rest, Sorted, SoFar) :-  /* The "SoFar" parameter is */
    split(Pivot, Rest, Lo, Hi),       /* used to avoid calling    */
    qsortx(Lo, Sorted, Pivot.SortHi), /* "append".                */
    qsortx(Hi, SortHi, SoFar).
split(Pivot, [], [], []).
split(Pivot, X.Rest, X.Lo, Hi) :- X =< Pivot,
                                  split(Pivot, Rest, Lo, Hi).
split(Pivot, X.Rest, Lo, X.Hi) :- X > Pivot,
                                  split(Pivot, Rest, Lo, Hi).
```

These predicates are all deterministic if their parameters are sufficiently instantiated. With the tail recursion optimization (TRO [Warren 1986]), `split` can be transformed into iterative form. Yet, the resulting code is still far from what a "conventional" programmer would produce because he (or she) would sort not a list but an array and the sorting would be done in-place. Lists should be sorted using other techniques such as merge sort (in fact, if the keys are expensive to compare, merge sort may be better even for arrays).

Here is *quick-sort* as it would be written for arrays.

```
qsort(Unsorted, Sorted) :-
    qsortx(Unsorted, {lob Unsorted}, {hib Unsorted}, Sorted).

qsortx(Sorted, Lo, Hi, Sorted) :- Lo >=  Hi.
qsortx(Unsorted, Lo, Hi, Sorted) :- Lo < Hi,
    Pivot = {Unsorted[Lo]},
    split(Unsorted, Lo, Hi, Pivot, PI, Split),
    qsortx2(Split, Lo, Hi, Lo, PI, Sorted).

qsortx2(Split, Lo, Hi, PI0, PI, Sorted) :- PI > Hi,
    /* Pivot value is >= everything; put at end and reduce range  */
    swapElem(Split, PI0, Hi, Split2),
    qsortx(Split2, Lo, {Hi-1}, Sorted).
qsortx2(Split, Lo, Hi, PI0, PI, Sorted) :- PI =< Hi,
    qsortx(Split, Lo, {PI-1}, Sort2),
    qsortx(Sort2, PI, Hi, Sorted).

split(Unsorted, Lo, Hi, Pivot, Lo, Unsorted) :- Lo > Hi.
split(Unsorted, Lo, Hi, Pivot, PI, Split) :-
    Lo =< Hi, {Unsorted[Lo]} =< Pivot,
    split(Unsorted, {Lo+1}, Hi, Pivot, PI, Split).
split(Unsorted, Lo, Hi, Pivot, PI, Split) :-
    Lo =< Hi, {Unsorted[Lo] > Pivot,
    split2(Unsorted, Lo, Hi, Pivot, PI, Split).

split2(Unsorted, Lo, Hi, Pivot, Lo, Unsorted) :- Lo > Hi.
split2(Unsorted, Lo, Hi, Pivot, PI, Split) :-
    Lo =< Hi, {Unsorted[Hi]} > Pivot,
    split2(Unsorted, Lo, {Hi-1}, Pivot, PI, Split).
split2(Unsorted, Lo, Hi, Pivot, PI, Split) :-
    Lo =< Hi, {Unsorted[Lo]} =< Hi,
    swapElem(Unsorted, Lo, Hi, Unsorted2),
    split(Unsorted2, Lo, Hi, Pivot, PI, Split).
```

This is a little longer than the algorithm for lists. It also implements a slightly different algorithm for `split` – the resulting split sub-arrays may have their elements in different orders than the sub-lists.

The array version can be easily modified to run more efficiently. A different method of choosing the pivot could pick a random member in constant time rather than the $O(\log N)$ time required for a list. Array lengths are always known, so a different, more efficient sorting method can be used when the number of elements in a sub-array becomes small.

It is too much to expect an optimizing compiler to transform the list formulation of *quick-sort* into the array formulation, especially as the two algorithms are slightly different. However, the compiler can easily determine that split is deterministic if the first parameter is instantiated, resulting in:

```
split(Unsorted, Lo, Hi, Pivot, Lo, Split) :-
  if Lo > Hi then Split = Unsorted
  elseif {Unsorted[Lo]} =< Pivot
    then split(Unsorted, {Lo+1}, Hi, Pivot, PI, Split).
    else split2(Unsorted, Lo, Hi, Pivot, PI, Split).
  endif.

split2(Unsorted, Lo, Hi, Pivot, Lo, Split) :-
  if Lo > Hi then Split = Unsorted
  elseif {Unsorted[Hi]} > Pivot
    then split2(Unsorted, Lo, {Hi-1}, Pivot, PI, Split)
  else
    swapElem(Unsorted, Lo, Hi, Unsorted2),
    split(Unsorted2, Lo, Hi, Pivot, PI, Split)
  endif.
```

which can then be transformed by the tail recursion optimization into the more conventional iterative form.

11.6 Direct access I/O

Some conventional programming languages treat direct access files like arrays. For example, XPL has the built-in pseudo-variable `file(i,j)` which maps to the `j`th record of the `i`th file. `X := file(i,j)` results in a record being read into `X`; `file(i,j) := X` results in a record being written from `X`.

The meaning of a program stays the same, regardless of whether the it uses direct access files or arrays; the sole difference is where the data are stored. A meta-predicate is used to associate a file with an array name and no change is required to the program. Copying files is even more expensive than copying arrays, so

```
?- ..., destructive(A, A2), ..., p(A), ...
```

always results in a run-time error when `A` is accessed within `p`. This is not a desirable situation. However, all implementations have limitations (for example, available memory or stack depth). This restriction preserves the correctness of programs – if they succeed or fail, they will do so exactly the same as programs without the restriction but sometimes they will neither succeed nor fail but will halt with an implementation restriction error indication.

Undoing changes to a direct access file on backtracking is relatively cheap; the mechanism is similar to the method used to undo changes to arrays.

The implementation of direct access files may impose other restrictions on their use. For example, there may be a requirement that all elements in a file be of the same type (this restriction can be relaxed with databases – see below). Also, files usually do not handle the concept of uninstantiated variables. These can be handled by keeping a separate list of output uninstantiated variables which get written when they become instantiated; a run-time error occurs if the program terminates with outstanding entries in this list.

11.7 Databases

A physical part of a database can be thought of as an array with non-integer indexes. The predicates which manipulate arrays can easily be extended to allow arbitrary indexes. Subrange predicates only make sense if the underlying index type can be ordered. Some additional predicates may be added, such as:

- find all elements based on some search criteria
- delete elements
- update elements
- sort

Using these predicates, a query language such as SQL can be easily implemented, if care is taken in avoiding the non-logical aspects of SQL.

Databases are related to associative table, as direct I/O is related to arrays. Databases usually contain some notion of "commit" processing. The changes to the database become permanent only when a specific "commit" transaction makes them permanent – up to that point, the changes can be "backed out." This fits very closely with Prolog's execution strategy. When there is the possibility of backtracking, "commit" can not be done. As soon as there are no backtrack possibilities for the predicate which modified the database (conceptually, produced a "new" database), "commit" can be done.

Conventional Prolog confuses the notion of databases by using the *assert* and *retract* predicates for both maintaining databases and for defining (or removing) predicates. Databases can be implemented – inefficiently – by ordinary lists or functors; the definition of predicates lies outside the scope of logical inference (see also section 10.7, "Assert and retract" on page 219).

11.8 Sequential Input/Output

[Cheng 1986] has observed that sequential I/O streams act just like lists (related ideas are given in [Wilson 1985]) Both must be processed element by element and both are of indeterminate length. Programs which do sequential I/O are written just as if they are manipulating ordinary lists. To make a program process stream files instead of lists, a meta-predicate is used to associate a file with a list. For input, whenever the unifier needs the head of this stream-list, it issues a file read. For output, whenever the unifier creates a new element in the list, it issues a file write.

Pascal uses a similar technique. A stream file is treated as a vector with a pointer to the current element. For logic programming, the stream pointer can be manipulated with `loRem` (to get individual elements from the beginning of an input stream) and `hiEx` (to add elements at the end of an output stream). The high bound of a stream is unavailable – any predicate requesting the high bound or length of a stream delays until end-of-file is reached.

Again, there is no difference between programs which do I/O and those which manipulate list. A meta-predicate declares that a particular list is associated with a file. For example:

```
?- instream('infile',  In),
   outstream('outfile', Out),
   qsort(In, Out).
```

will sort file "infile" into file "outfile" (this requires the list version of *quick-sort* rather than the array version).

For convenience, the various standard files can be automatically added to predicates.

```
p(A, B) :- put('A = '), display(A),
           q(A, B),
           put(' => '), display(B), nl.
q(A, B) :- put('<qqq>'), B is A + 1.
?- p(1,2).
```

would be transformed to

```
p(A, B, StdIn, StdOutRest, StdIn2, StdOut) :-
    StdOut = ('A = ' . A . StdOut2),
    q(A, B, StdIn, StdOut3, StdIn2, StdOut2),
    stdOut3 = (' => ' . B . '\n' . StdOutRest).
q(A, B, StdIn, StdOut, StdIn, StdOut2) :-
    StdOut2 = ('<qqq>' . StdOut),
    B is A + 1
?- instream('<terminal>', In), outstream('<terminal>', Out),
   p(1,2, In, Out, In2, []).
```

This is similar to techniques used to transform grammar rules to Prolog.

As with direct access files, I/O streams may impose some extra restrictions over lists. Because output may be to a non-erasable medium like a printer, an output stream has a buffer associated with it; backtracking can only undo values which are still in the buffer. The flush meta-predicate flushes everything in the buffer to the output device — a run-time error occurs if an attempt is made to flush an uninstantiated variable. Flush does not change the meaning of a program, nor the final contents of a file, but it may cause a run-time error to occur in an otherwise correct program.

Input and output with streams is very similar to lazy evaluation. An input file is read only as far as needed and is automatically closed when the program terminates. An output file is closed by instantiating it to []. When a program ends, all stream files are closed and the buffers flushed.

The following program will copy infile to outfile:

```
?- instream('infile',  In),
   instream('outfile', Out),
   In = Out.
```

Two input files are compared by:

```
?- instream('infile1', In1),
   instream('infile2', In2),
   In1 = In2.
```

In both these examples, the "=" can be done away with:

```
?- instream('infile,'  CopyFile),
   instream('outfile', CopyFile).

?- instream('infile1', CompareFile),
   instream('infile2', CompareFile).
```

It should be noted that streams can be added to a logic programming language which implements delays. For example, the query

```
?- outstream('file2', Out), query(In, Out), instream('file1', In).
```

reads in `file1` and passes it through `query` to write out `file2`. The order of the goals are important to ensure lazy evaluation – the order does not affect the meaning of the query. The stream predicates can be written:

```
instream(File, X.Rest) :- readFile(File, X), inStream(Rest).
instream(File, [])     :- reachedEof(File).

outstream(File, X?.Rest) :- writeFile(File, X), outStream(Rest).
outstream(File, [])      :- closeOutput(File).
```

where `readFile`, `reachedEof`, `writeFile` and `closeOutput` are non-logical I/O predicates as in most Prologs. However, streams should be integrated into the unification mechanism or treated as lazy functions (see section 6.4.1, "Lazy thunks vs. delayed predicates" on page 132).

Many varieties of the *stream* meta-predicate are possible. For example, we might want streams which pass single characters or strings (delimited by white space or delimiters) or even general purpose pattern matching, like that provided by the C library routine `scanf`.

11.9 Implementation of many object types

The above discussion has proposed adding some new objects to a logic programming implementation, including

- arrays

- arrays with reference count of 1
- strings
- strings with reference count of 1
- sub-arrays (and sub-strings)
- direct-access files
- databases
- I/O streams

All these objects have a similar appearance within programs but have markedly different implementations. Their various internal representations are invisible to the Prolog programmer.

Many logic programming implementations provide some form of clause indexing. For example, the WAM provides a *switch-on-term* which causes a jump to one of a number instructions depending on whether the object is a variable, nil, list element, atom or structure (WAM restricts this to the first parameter of a predicate although there is no need for that restriction (see 3.0, "The basic sequential inference engine" on page 28)). This instruction can be easily extended to include any number of other object types.

For arrays, the list expression `A = Hd.Tl` can be. interpreted as `{len(A)} > 0, loRem(A, Hd, Tl)`. For termination, `[]` unifies with a 0-lengh array. The Prolog code

```
p([], []).
p(Hd.Tl, Hd2.Tl2) :- q(Hd, Hd2),
                     p(Tl, Tl2).
```

can also be interpreted as

```
p(A, A2) :- {len(A)} = 0,
            bounds(A2, 0, -1). /* nil array */
p(A, A2) :- {len(A)} > 0,
            loRem(A, Hd, Tl),
            q(Hd, Hd2),
            p(Tl, Tl2),
            hiEx(Hd2, Tl2, A2).
```

resulting in code something like this (in pseudo-C):

```
p(parm1, parm2) { /* general case for any types of parms */
    again:
    switch (typeTag(parm1)) {
    case typeNil:
        unifyWithNil(parm2);
        break;
    case typeListElement:
        unifyListElement(parm1, hd,  tl);
        unifyListElement(parm2, hd2, tl2);
        q(hd, hd2);
        parm1 = tl;  parm2 = tl2;
        goto again; /* tail recursion => iteration */
    case typeStream:
        readFromStream(parm1, hd, tl);
        /* ... continue as for typeListElement ... */
        goto again; /* tail recursion => iteration */
    case typeArray:
        if (refCount(parm1)) > 1) {
            a = copyArray(parm1);
            pArray(a); /* deterministic update (code is below) */
            unify(a, parm2);
        } else {
            pArray(parm1);
            unify(parm1, parm2);
        }
        break;

    ... cases for other types ...
    default:
        error(...);
    }
}

void pArray(a) { /* destructive updating version for arrays */
    for (i = lob(a); i < hib(a); i++) {
        q(a[i], ax);
        a[i] = ax;
    }
}
```

Any number of new internal types can be added to this skeleton. Of course, the amount of generated code will increase but execution speed will remain the same. If the logic programming language has a good optimizer, some of the cases in the above code can be determined at compile time.

11.10 Expressiveness

Many logic programs suffer from the "if you have a hammer, everything looks like a nail" syndrome. The main tool for structuring repetitive data is the list, so programs are written using lists, whether they are appropriate or not. A similar problem often arises in Fortran – structures are smashed into arrays because they are the only structuring tools in the language. For a partial discussion of one aspect of this problem, comparing records (functors) versus lists and selector functions, see [Wadler, 1987].

Consider a predicate which looks up a name N in a table T and returns its index Index (starting from 1). If the name is not in the table, it is added to the end, giving the new table NewT.

```
lookup(N, T, NewT, Index) :- lookup2(N, T, 1, NewT, Index).
lookup2(N, [], Index, [N], Index).
lookup2(N, N.Rest, Index, N.Rest, Index).
lookup2(N, N2.Rest, I, N2.Rest2, Index) :- N ≠ N2,
    lookup2(N, Rest, {I + 1}, Rest2, Index).
```

Here is array based code:

```
lookup(N, T, NewT, Index) :- lookup2(N, T, 1, NewT, Index).
lookup2(N, T, Index, NewT, Index) :- Index > {hib T},
                                     hiEx(T, N, NewT). /* NewT = T || N */
lookup2(N, T, Index, T, Index) :- Index =< {hib T},
                                     {T[Index]} = N.
lookup2(N, T, I, NewT, Index) :- Index =< {hib T},
                                   {T[I]} ≠ N,
                                   lookup2(N, T, {I + 1}, NewT, Index).
```

Even though it is slightly longer, the array version is clearer because it says directly that the Indexth element of T is N — this must be deduced by an inductive proof for the list version. Furthermore, the array version creates a new table only if the name N is not already in the old table (which can often be done by destructive assignment to the old table). If the list version were modified to create a new list only when the name is not in the old list, it would be longer and much less clear than the array version.[30] If greater efficiency is desired, the list version can be transformed to use a binary tree. But the

[30] Some Prolog textbooks have truly horrible examples, using a list with an uninstantiated variable as the tail, then using *var* to determine whether or not the name is in the list. This

greatest efficiency is attained with a hash table which is efficient only because all elements are in an array and are accessible in constant time.

Arrays allow symmetric access to elements. An algorithm can as easily be written to process from the end of an array as from the beginning. Lists are best handled only from the beginning to the end. Some people would claim that logic programmers do not write algorithms, just specifications. But there is some concern for efficiency, otherwise we could just write:[31]

```
sort(Unsorted, Sorted) :- permutation(Unsorted, Sorted),
                          ordered(Sorted).
```

Especially in "very high level" languages, programs often do not deal with lists or arrays but with sets or multi-sets, mappings and similar mathematical objects. These can be realized as lists or arrays but often they are better handled as abstract types. For example, the array handling predicates could be extended to work with lists (although, in this case, very inefficiently):

var technique does not work for more complex problems, such as are encountered by register allocation algorithms.

31 Although with all the clauses for `ordered` and `permutation`, this is actually longer than *quick-sort*.

```
elem(L, I, E) :- islist(L), elemOfList(L, 0, I, E).
/* treat a list as a 0-origin array: */
elemOfList(E.Rest, I, I, E).
elemOfList(E.Rest, Lo, I, E) :- Lo < I,
                                elemOfList(Rest, {Lo+1}, I, E).

chElem(L, I, E, LOut) :- isList(L), chElemOfList(L, 0, I, E, LOut).
chElemOfList(X.Rest, I, I, E, E.Rest).
chElemOfList(X.Rest, Lo, I, E, X.RestOut) :- Lo < I,
    chElemOfList(Rest, {Lo+1}, I, E, RestOut).

swapElem(A, I, J, AOut) <- isList(A),
    elemOfList(A, I, AI) &
    elemOfList(A, J, AJ) &
    chElemOfList(A, I, AJ, A2) &
    chElemOfList(A2, J, AI, AOut).
```

No one would want to use such predicates for quick-sort but they will do the job correctly. It is not unreasonable to build knowledge of such predicates into a Prolog compiler, so that the appropriate data conversions are done when needed. Thus, if the programmer entered

```
?-qsort(List, OutputList), q(OutputList).
```

where *List* and *OutputList* are lists, the compiler would convert this to:

```
?-listToArray(List, Array), qsort(Array, Array2),
  arrayToList(Array2, OutputList), q(OutputList).
```

by noticing that *qsort*'s preferred data type is an array. Such data transformations have been investigated for a long time, for example in [Schwartz 1975]. Pure logic programming allows such a style of programming with optimizations being applied by a combination of automatic type inferencing and dialogue with the programmer.

FP [Backus 1978] and APL treat arrays as simple objects. Implementing these arrays requires optimizations similar to those given above.

FP depends heavily on functions which manipulate other functions. In logic programs, we can easily have predicates which manipulate other predicates (see 6.4, "Thunks, lazy evaluation and higher order functions" on page 127). Although predicates can be treated as "first class objects," they cannot be unified in the same way as other objects – there is no way to tell if arbitrary recursive predicates are the same.

`Assert` and `retract`[32] have two uses: to dynamically add new clauses and to implement data bases. The former use can be problematic if the logic program is compiled; the latter use can be handled better by the array or database predicates described above.

Non-logical I/O is much less expressive than logical I/O. It requires that goals be in a particular order (contrary to the commutative nature of *and* (",")). Non-logical I/O does not easily support backtracking.

32 addax and delax in Waterloo and IBM Prologs.

Conclusion

Logic programming is still in its infancy. Just as Fortran was superseded by Algol, PL/I, Pascal, Ada and many others, Prolog will surely be superseded. Prolog is a first attempt at a new style of programming. As such, we should not criticize its defects, but applaud its boldness. We can only hope that its defects do not linger on, as do Fortran's. Innovations are needed.

Logic Programming is now well-established as a practical method of building computer systems. My own experience, and that of many others, is that logic programming can give programmer productivity increases of 10 – 20 times. The challenge is to make logic programs run as fast as those produced by conventional means, and to provide tools for building large logic programs.

I have explored a variation on the popular WAM implementation of a logic engine (**xpPAM**) which retains WAM's efficiency, yet implements a more powerful language than conventional Prolog, providing sound negation and coroutining. This design is also suitable for functional programming.

I have implemented the logic engine (except for virtual memory), including an prototype optimizing compiler, and have attained performance comparable to WAM implementations. The compiler is short and simple because the abstract logic engine's instructions lend themselves to easy compilation. Suitable language constructs – such as delays, *if-then-else* and meta variables – both help the compiler and allow programmers to write clearer code.

Logic programming is a superior paradigm for many problems, compared to conventional programming techniques, because it improves programmer productivity, allows rapid prototyping and encourages writing provably correct programs. My work shows that logic programs can run as fast as conventional programs, without any need to introduce "impure" non-logical features.

Logic programming is a superior paradigm for many problems, compared to conventional programming techniques – it improves programmer productivity, allows rapid prototyping and encourages writing provably correct programs. My work shows that pure logic programs can be executed efficiently, with any need to introduce "impure" non-logical features.

Continued work on techniques for compiling logic programs, such as those in this report, will give pure logic programs the efficiency of conventional programs.

"If anybody wants to clap," said Eeyore when he

had read this, "now is the time to do it."

They all clapped.

"Thank you," said Eeyore. "Unexpected and
gratifying if a little lacking in Smack."

— A. A. Milne, *Winnie the Pooh*

Glossary and Index

address The index into memory of an object. See also “pointer,” section3.1, “Objects” on page 29 .

algorithm A method of computing a proof. Kowalski's definition is: *algorithm* = *logic* + *control.*

all-solutions predicate A second-order predicate which computes all the solutions for a predicate. These are usually called *bagof* and *setof*. *Bagoff* is in computed order and may contain duplicates; *setof* is ordered and contains no duplicates. In some implementations, no solution results in *nil*; in other implementations, the all-solutions predicate fails. See section 8.9, “All solutions predicates” on page 178.

argument A value passed to a goal. See also “parameter.”

atom	A simple value, typically a name (string) or a number. See also "compound term."
axiom	A predicate or clause.
backtrack	To reset the machine to an earlier state and pick an alternate clause to resume computation. See also "non-deterministic," section 4.0, "Backtracking and delaying" on page 66.
backtrack stack	A stack which records information necessary to reset the machine to an earlier state. Also called "choice stack" or "choice point stack" See section 4.0, "Backtracking and delaying" on page 66.
call	Attempt to compute a goal (also, "try" a goal).
choice point	A collection of information in the backtrack stack which contains information about an alternative clause to take on backtracking. See section 4.0, "Backtracking and delaying" on page 66.
clause	One alternative in a predicate. It consists of a "head" and zero or more "goals" which must all be satisfied if the clause is to be satisfied. See section 8.2, "Conventional Prolog's execution order" on page 159.

closed (completed) predicate A predicate which has all its alternatives given. Its negative can therefore be computed with the "closed world" technique. See section 10.1, "Negation" on page 207.

code segment In xpPAM, the abstract machine instructions, plus constants for one predicate. A code segment may also contain instructions for predicates which are strictly internal to it. See section 3.6, "Code segments" on page 36.

complete Describing a proof procedure which will always terminate successfully if the goal is provable. See also "sound."

complex indeterminate A compound term which contains field (slot) names and associated values. Has possibilities for implementing frames (q.v., second meaning). See section 3.14, "Other object data types" on page 59.

compound term A term of the form $f(t_1, t_2, \dots t_n)$ where each t_i may be either an atom or a compound term. Also called a "functor." Some compound terms have special forms. For example, a list element is notated "*A.B*," "[*A*|*B*]" or "*.(A,B)*" – in all cases, the functor name is "." and the arguments are *A* and *B*.

computation A series of steps that a machine takes in attempting to prove a query. See also "proof procedure."

coroutine	A predicate which may suspend before its computation is complete. Typically, one or more predicates cooperate by suspending and resuming. Coroutining may be considered as a form of parallelism. See section 9.0, "Coroutining, pseudo-parallelism and parallelism" on page 185.
cut ("!")	An "impure" predicate in conventional Prolog which "cuts" the solution space. Its declarative reading is "true" but it can change the generated solutions. Cuts can be avoided by use of suitable constructs (such as *if-then-else*) or by proper implementation (tail recursion optimization, etc.).
declarative reading	The logical reading of a Prolog predicate. If the predicate contains only purely logical predicates, the declarative reading can be produced by "or"-ing the clauses, replacing commas by "and"s and adding existential and universal quantifiers. If the predicate contains non-logical predicates (*var*, *cut* ("!"), I/O, etc.), a declarative reading is much more difficult.
delay	Postpone evaluation of a predicate until one or more arguments become sufficiently instantiated. See sections 1.5, "Delay notation" on page 8 and 8.10, "Non-strict execution order: delays" on page 181.
deterministic	A computation which can only proceed in on sequence of steps, producing a single answer.

deterministic append A predicate which appends two lists to make a third list. This is the inner loop of "naïve reverse." *Append* can also be non-deterministic, in which case it is used to split a list into two.

```
append([], X, X).
append(X.A, B, X.C) :- append(A, B, C).
```

execution stack A stack which keeps information across calls. Also called "call stack." See section3.5, "Execution stack" on page 34 .

fact A clause (or predicate) which contains no goals and completely grounds its arguments. See also "rule."

fail In a computation, if a predicate does not succeed, it is said to fail. If a goal fails and the computation is sound, then there is no possible proof for the goal.

frame Two meanings:

- A collection of related information on the execution or backtrack stack.
- A collection of information associated with an object (used for AI or object-oriented programming). Typically, the frame slots are values or procedures for computing the values. See also "complex indeterminates."

free list A list of cells which are not yet allocated to objects. See also section 7.5, "Allocating from a list or from a stack" on page 145.

freeze (or *geler***)** The predicate in Colmerauer's Prolog-II which implements "delay." See also section9.7, "Comparison with other designs for delaying." on page 200 .

function A mapping from one set of objects to another. An *n*-argument function can be transformed to an *(n + 1)*-argument deterministic predicate by "returning" the answer in the *(n + 1)*th argument.

functor A compound term.

fully instantiated (or ground) Has a value with no uninstantiated variables. See "instantiated" and "sufficiently instantiated."

functor element cell If a functor is represented internally as a list of cells, this is like a list element cell but with a special flag so that it will print out in functor notation and will only unify with another functor. See also "list element cell," sections 3.1, "Objects" on page 29, 7.10, "Functor and list storage" on page 151.

goal One of the terms on the right hand side of a clause which must be satisfied if the clause is to be satisfied. See section 8.2, "Conventional Prolog's execution order" on page 159.

ground	Has a value. See "instantiated."
head	The name and parameters of a clause.
heap	An area of memory within which objects may be allocated and freed in any order. See also "stack," section 7.5, "Allocating from a list or from a stack" on page 145.
Horn clause	A clause containing a head and zero or more goals. The more general form (from Gentzen's sequent calculus) allows more than one term in the head.
KLIPS	Thousand LIPS.

if-then-else A control construct which can be defined:

```
p :- if t then q else r.
```

is the same as:

```
p :- t, q.
p :- not t, q.
```

See section 8.6, "Single solutions" on page 170.

impure predicate A predicate which cannot easily be given a logical or declarative meaning. For example, *cut* ("!") and *var* require knowledge of the computational mechanism; I/O predicates

	require transformations (with extra arguments) to be given declarative meanings.
instantiate	To cause an uninstantiated variable object to receive a value. This normally happens during unification. See section 3.10, "Unification" on page 40.
instantiated	Has a value. Also called "ground." When an object is created during computation, it is initially uninstantiated; it may become instantiated during unification. Once instantiated, an object cannot change its value; backtracking may cause it to become uninstantiated again.
LIPS	Logical inferences per second. Usually computed with the "naïve reverse" benchmark which typically gives speeds about three times faster than "typical" predicates. There are rumours that some implementations have a built-in "naïve reverse" instruction (or deterministic append instruction) to give high LIPS figures. See section 5.9, "Optimized compiling to conventional machine code" on page 110.
list element cell	An object containing a "head" and a "tail" (corresponding to LISP's *car* and *cdr*). For example, the list "[*A*,*B*]" contains two list element cells: the first's head is *A*, the first's tail points at the second cell; the second's head is *B* and the second's tail contains *nil.* See also "functor," sections 3.1, "Objects" on page 29, 7.10, "Functor and list storage" on page 151.

logical predicate A "pure" predicate which can easily be given a logical or declarative meaning in first (or second) order predicate calculus.

meta-predicate A predicate which has no effect on correctness of a computation but which may control some aspect of the representation or of the order of computation. For example, a meta-predicate may supply delaying information or it may associate an input list with a file. Sometimes, "second order" predicates (such as *call* are called meta-predicates. See section 11.8, "Sequential Input/Output" on page 241 for examples.

meta-variable A variable which is used as a "place holder" within a predicate skeleton. This is typically used for all-solutions predicates or for knowledge representation which uses predicate forms. See section 10.4, "Meta-variables" on page 212.

MIPS Million instructions per second (also called "meaningless indicator of processor speed").

MLIPS Million LIPS.

naïve reverse A predicate which reverses a list using a rather simple-minded algorithm which mainly consists of calls to deterministic append. Naïve reverse is often used to give LIPS figures.

```
naive_reverse([], []).
naive_reverse(X.A, R) :- naive_reverse(A, A2),
                         append(A2, [X], R).
```

negation The contra-positive of a predicate. That is, if a predicate evaluates to "true," its negation evaluates to "false." Some logic systems allow making negative statements; others, such as Horn clause logic allow only making positive statements. There are a number of kinds of negation, including:

classical negation which requires full resolution in the general case (for example, given "all politicians exaggerate" and "John does not exaggerate," this could derive "John is not a politician").

closed world negation which will only correctly negate full ground predicates by checking if they are not in the database of clauses. It is simple to compute but less complete that classical negation

See sections 10.1, "Negation" on page 207, 8.5, "Negation" on page 168.

non-deterministic A computation in which the order is not known ahead of time. There are two main kinds of non-determinism: order of clause selection and multiple answers.

non-logical Not capable of being given an simple first or second order predicate calculus meaning. See "impure predicate."

nonvar	An "impure" predicate in conventional Prolog which succeeds if its argument is instantiated. It has no possible declarative reading. *Nonvar* can be avoided by use of suitable control constructs (such as "delay").
object	A piece of allocated memory with a tag indicating its type. See section 3.1, "Objects" on page 29.

parallel computation A computation in which more than one predicate are computed in parallel. There are two main forms:

and-parallelism requires all the parallel predicates to succeed; it waits until the last predicate succeeds and is typically used for divide-and-conquer algorithms.

or-parallelism requires one of the parallel predicates to succeed; it is typically used for database searches.

parameter	A value passed into a clause. See also "argument."
pointer	An object which contains the address of an object. See also "address," section3.1, "Objects" on page 29 .
predicate	A term of the form $p(f_1, f_2, \ldots, f_n)$ which is supposed to have a value of "true" or "false" for each instantiated substitution for its variables. A predicate may be (recursively) computed, so some substitutions may not be computable.

Prolog	"Programming in Logic": a logic programming language, originating from the work of Colmerauer and Kowalski. It has a number of dialects, of which the most popular is Edinburgh (DEC-10) Prolog and its derivatives (C-Prolog, etc.). IBM Prolog uses a somewhat different syntax which can (usually) be mechanically transformed to Edinburgh Prolog. Most of the differences among Prolog dialects are minor, usually in details of built-in predicates, especially "impure" predicates.
proof procedure	A mechanical method of proving a goal. Sometimes called a computational method. A proof procedure typically selects some path through the tree of all possible proofs. See section 8.3, "Example of conventional execution order" on page 163.
pure	Containing only logical predicates. "Pure Prolog" is sometimes, mistakenly, referred to as a subset of Prolog.
query	A goal or conjunction of goals which the machine will attempt to prove (in the original theory, a query is a single negated goal and the computation attempts to find a substitution which results in a contradiction; this corresponds to finding a substitution which satisfies the axioms and original goal).
register	In xpPAM, a special piece of memory containing the address of an object. A register may be empty, in which case its contents are meaningless. See section 3.2, "General registers" on page 32.

The status of the machine is kept in special status registers, see section 3.4, "Status registers" on page 33.

register annotation One of the letters *v*, *n*, *f*, *x* or *s* which gives information about the contents of a register and what is to be done with it in an instruction. The annotation *c* indicates that the operand is not a register but an index into the constants' vector. See section 3.7, "Machine instruction format" on page 36.

reset stack	A stack which contains information about variables which have become instantiated and which require being "un-instantiated" on backtracking. Also called "trail." See section 4.0, "Backtracking and delaying" on page 66.
resolution	A method discovered by Robinson based on Gentzen's cut rule which provides a single rule for doing a proof. Full general resolution can have exponential cost.
resume	To continue execution of a predicate where is was earlier suspended. See also "suspend," "coroutine."
rule	Sometimes used to mean "clause" or "predicate." A rule normally has at least one goal. See also "fact."
satisfy	To try a goal and succeed or to find a substitution during a unification.

sequent calculus	A method of proving first order predicate calculus goals, using an alternate notation which was invented by Gentzen. Horn logic is a subset of sequent calculus.
sound	Describing a proof procedure which always gives a correct answer (success or failure corresponding to provable or unprovable). See also "complete."
stack	An area of memory from which objects may be allocated and freed in strict LIFO order. See also "heap," section 7.5, "Allocating from a list or from a stack" on page 145.
string	Zero or more characters. If the string starts with a lower case letter and contains only printable non-space characters, it can be written without enclosing quotes, like an atom. Otherwise the quotes are not needed: "`abc='abc'`."
substitution	A set of values which satisfy some equalities (usually, computed by unification).
succeed	Of a predicate, to be proved. See also "fail."
suspend	To cease execution of a predicate and start or resume execution of another before the current predicate has terminated with either success or failure. See also "resume," "coroutine."

tail recursion optimization An optimization which turns the call to the last goal in a clause into a kind of *go to*. Also called "TRO." TRO is used to transform recursion into iteration. See section 3.9, "Tail recursion optimization (TRO)" on page 39 and the "last call" instructions.

term-rewriting A general proof procedure (used, for example, by sequent calculus) in which an arbitrary goal is chosen and its right hand side is substituted for it. See section 8.4, "A more flexible execution strategy" on page 165. Similar to Markov algorithm. If the left-most goal is always chosen, term-rewriting is the same as conventional Prolog's depth-first left-to-right computation rule.

TRO Tail recursion optimization.

try Attempt to compute a goal (also, "call" a goal).

thunk An object which contains a predicate name and some or all of the arguments for it. If all the arguments are present, it is very similar to a suspended predicate. See section 6.4, "Thunks, lazy evaluation and higher order functions" on page 127.

trail WAM terminology for the reset stack.

sufficiently instantiated (or ground)	Has enough of a value for computation to proceed. For example, the computation may require that an object be instantiated to a list element, but the head or tail may still be uninstantiated. See also "instantiated" and "delay."
unification	The process whereby two terms (possibly containing uninstantiated variables) are made "equal." Unification may fail. Unification may cause variables to become instantiated, always to the "most general unifier." For example, "`X=Y,Y=a`" results in `X=a` and `Y=a`; "`X=Y,Y=Z`" results in `X=Y`, `X=Z` and `Y=Z` (the most general unifier) even though `X=Y=Z=a` will satisfy the equation. See section 3.10, "Unification" on page 40.
var	An "impure" predicate in conventional Prolog which succeeds if its argument is uninstantiated. It has no possible declarative reading. *Var* can be avoided by use of suitable control constructs (such as "delay").
variable	Two meanings: • a name in a predicate, usually starting with a capital letter. • an uninstantiated variable during the course of a computation.
WAM	Warren Abstract Machine.

Warren Abstract Machine	A design by D. H. D. Warren for an abstract machine which has proven to be very efficient. It is the basis of most recent hardware and software implementations of Prolog. See section 7.1, "Comparison with the Warren Abstract Machine instructions" on page 138.
weak delay	A delay which has an associated "cost." Unlike a regular delay, a weak delay may resume without its argument(s) being instantiated, but the computation is likely to be expensive (the cost gives an indication of the expense). See section 4.6, "Weak delays: dynamic reordering of clauses" on page 81.
`xpPAM`	The `xpProlog` abstract machine. An abstract machine design suitable for implementing conventional Prolog, `xpProlog` or functional languages. It has some similarities with WAM but is both more simple and more flexible, while retaining WAM's efficiency. `XpPAM` can be interpreted on conventional machines, used an intermediate language when compiling to conventional machines, or `xpPAM` could be implemented in hardware. See sections 3.0, "The basic sequential inference engine" on page 28. and 4.0, "Backtracking and delaying" on page 66.
`xpProlog`	*Extended pure Prolog.* The pure "subset" of Prolog, extended with constructs (such as *if-then-else*, meta-variables, delays, etc.) which make "impure" predicates unnecessary.

References

Abramson, H. [1984] *A Prological Definition of HASL a Purely Functional Language with Unification Based Conditional Binding Expressions.* New Generation Computing, 2. Springer-Verlag.

Auslander, M. and Hopkins, M [1982] *An Overview of the PL.8 Compiler.* Proc. SIGPLAN 1982 Symposium on Compiler Construction.

Backus, J. [1978] *Can programming be liberated from the von Neumann style? (A.C.M. Turing Award Lecture).* Communications of the A.C.M., 21(8).

Bosco, P. and Giovanetti, E. [1986] *IDEAL: An Ideal DEductive Applicative Language.* Proc. IEEE 1986 Symposium on Logic Programming.

Bratko, I. [1986] *Prolog Programming for Artificial Intelligence.* Addison-Wesley 1986

Bruynooghe, M. [1982] *The Memory Management of Prolog Implementations.* In "Logic Programming": Clark, K.L., Tärnlund, S-A. (ed.). Academic Press.

Bruynooghe, M. [1986] *Compile time garbage collection* Report CW43, Department Computerwetenschappen, Katholieke Universiteit Leuven.

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

Campbell, J. and Hardy, S. [1984] *Should Prolog be list or record oriented?.* In "Implementations of Prolog": Campbell, J. (ed.). Ellis Horwood (1984).

Cheng, M. [1986] *Logical I/O for Prolog.* Dept. of Computer Science, University of Waterloo.

Clark, K. [1978] *Negation as failure.* In "Logic and Databases": Gallaire, H. and Minker, J. (ed.). Plenum Press.

Clark, K. and Gregory, S. [1984] *PARLOG: Parallel Programming in Logic.* Research report DCO 84/4, Dept. of Computing, Imperial College, London. Also in A.C.M. Transactions on Programming Languages and Systems 8(1) (January 1986).

Clark K., McCabe, F. and Gregory, S. [1982] *IC-Prolog Language Features.* In "Logic Programming": Clark, K.L., Tärnlund, S-A. (ed.). Academic Press.

Clark, K. and McCabe, F. [1984] *micro-PROLOG: Programming in Logic.* Prentice-Hall.

Clocksin, W. and Mellish, C. [1981] *Programming in Prolog.* Springer-Verlag.

Cohen, P and Feigenbaum, E. (ed.) [1982] *The Handbook of Artificial Intelligence, Vol. 3.* HeurisTech Press.

Colmerauer, A. [1982] *PROLOG-II Manuel de Référence et Modèle Théorique,* Groupe Intelligence Artificielle, Univ. d'Aix-Marseille II.

Dahl, O-J., Dijkstra, E. W. and Hoare, C. A. R. [1972] *Structured Programming.* Academic Press.

Debray, S. [1985] *Register allocation in a Prolog machine.* Technical Report 85/10, State University of New York, Stony Brook.

Debray, S. [1986] *Towards Banishing the Cut from Prolog.* Proc. IEEE 1986 International Conference on Computer Languages.

Debray, S. and Warren, D. S. [1986] *Detection and Optimization of Functional Computations in Prolog.* Proc. Third International Conference on Logic Programming. Springer-Verlag 1986.

DeGroot, D. and Lindstrom, G. [1986] *Logic Programming Functions, Relations and Equations.* Prentice-Hall.

Dijkstra, E. W. [1968] *Go to statement considered harmful.* Communications of the A.C.M. 11 (March 1986).

Dijkstra, E. W. [1976] *A Discipline of Programming.* Prentice-Hall.

Dobry, T. P. [1987] *A high performance architecture for Prolog.* Ph. D. thesis, report UCB/CSD 87/352, University of California at Berkeley.

Dobry, T. P., Patt, Y. N. and Despain, A. M. [1984] *Design decisions influencing the microarchitecture for a Prolog machine.* Micro 17 Proceedings, October 1984.

Gabriel, Linkholm, Lusk and Overbeek [1985] *A Tutorial on the Warren Abstract Machine for Computational Logic.* Argonne National Laboratory Report ANL-84-84.

Gabriel, R. P. [1985] *Performance and Evaluation of Lisp Systems.* MIT Press.

Golderg, A. [1983] *Smalltalk-80: The Language and its Implementation.* Addison-Wesley.

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

Hermenegildo, M. and Nasr, R. [1986] *Efficient Management of Backtracking in AND-Parallelism.* Proc. Third International Conference on Logic Programming. Springer-Verlag 1986.

Hoare, C. A. R. [1986] *Communicating Sequential Processes.* Prentice-Hall.

Hodges, W. [1971] *Logic.* Penguin Books.

Ingerman, P. [1961] *Thunks - A way of compiling procedure statements with some comments on procedure declarations.* Communications of the A.C.M. 4, 1.

Kleene, S. [1967] *Mathematical Logic.* John Wiley & Sons.

Kluzniak, F. [1981] *Remarks on Coroutines in Prolog* In "Papers in Logic Programming I.":, Report 104, University of Warsaw, Institute of Informatics (also for the closed Workshop on Logic Programming for Intelligent Systems, 18-21 August 1981, Long Beach Harbor, California).

Kluzniak, F. and Szpakowicz, S. [1985] *Prolog for Programmers.* Academic Press.

Kluzniak, F. [1987] *Compile time garbage collection for ground Prolog* (unpublished) Warsaw University Institute of Informatics.

Knuth, D. E. [1973] *The Art of Computer Programming, Vol. 1: Fundamental Algorithms*, pp. 417-420. Addison-Wesley.

Kowalski, R. [1979] *Logic for Problem Solving.* Elsevier North Holland

Kowalski, R. [1979b] *Algorithm = Logic + Control.* Communications of the A.C.M. August 1979

Krasner, G. (ed.) [1983] *Smalltalk-80: Bits of History, Words of Advice.* Addison-Wesley.

Jaffar, J. and Lassez, J.-L. [1987] *Constraint Logic Programming.* Proc. Conference on Principles of Programming Languages.

Landin, P. [1966] *An abstract machine for designers of computing languages.* Proc. IFIP Congress 65, Vol. 2, Washington. Spartan Books.

Lloyd, J. [1984] *Foundations of Logic Programming.* Springer-Verlag.

Lloyd, J. and Topor, R. [1984] *Making Prolog more Expressive.* The Journal of Logic Programming, 4.

Matsumoto, H. [1985] *A Static Analysis of Prolog Programs.* SIGPLAN Notices, V20 #10, October 1985.

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

McKeeman, W., Horning, J. and Wortman, D. [1970] *A Compiler Generator implemented for the IBM System/360.* Prentice-Hall.

Mills, J. W. [1986] *A high performance LOW RISC machine for logic programming.* Proc. IEEE 1986 3rd International Symposium on Logic Programming.

Moss, C. [1986] *CUT & PASTE - defining the impure Primitives of Prolog.* Proc. Third International Conference on Logic Programming. Springer-Verlag 1986.

Mukai, K. and Yasukawa, H. [1985] *Complex Indeterminates in Prolog and its Application to Discourse Models.* New Generation Computing, 3.

Naish, L. [1985a] *Automating Control for Logic Programs.* The Journal of Logic Programming, Vol. 2, Num. 3, October 1985.

Naish, L. [1985b] *Negation and Control in Prolog.* Ph. D. Thesis, University of Melbourne. Also available as *Springer-Verlag Lecture Notes in Computer Science #238* (Goos, G. and Hartmanis, J., eds.) (1986).

Naish, L. [1985c] *All solutions Predicates in Prolog.* Proc. IEEE Symposium on Logic Programming (July 1985).

Naish, L. &lbrk1985d] *The MU-Prolog 3.2 Reference Manual.* Department of Computer Science, University of Melbourne

Naish, L. [1986] *Negation and quantifiers in NU-Prolog.* Proc. Third International Conference on Logic Programming. Springer-Verlag 1986.

O'Keefe, R. [1985] *On the Treatment of Cuts in Prolog Source-Level Tools.* Proc. 1985 Symposium on Logic Programming.

Periera, F., Warren, D. H. D., Byrd, L. and Pereira, L.M. [1984] *C-Prolog User's Manual Version 1.5.* SRI International, Menlo Park, California.

Pirsig, R. [1974] *Zen and the Art of Motorcycle Maintenance.* William Morrow (also Bantam Books).

Plaisted, D. [1984] *The Occurs-check Problem in Prolog.* Proc. IEEE 1984 International Symposium on Logic Programming.

Quine, W. [**1941, revised 1965**] *Elementary Logic.* Harper and Row.

Richards, M and Whitby-Stevens, C. [1979] *BCPL - the language and its compiler.* Cambridge University Press.

Sahlin, D [1986] *Making tests deterministic using the reset information.* SICS, Sweden.

Schwartz, J. [1975] *On Programming: an interim report on the SETL project.* Courant Institute of Mathematical Sciences, New York University.

Sergot, H. [1983] *A Query-the-User facility for logic programming.* Proc. European Conference on Integrated Interactive Computer Systems: Degano, P. and Sandewall, E., eds. North-Holland.

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

Sterling, L. and Shapiro, E. [1986] *The Art of Prolog.* The MIT Press.

Tick, E. [1985] *Prolog Memory-Referencing Behavior.* Stanford University Technical Report No. 85-281 (September 1985)

Tick, E. [1986] *Memory performance of Lisp and Prolog programs.* Proc. Third International Conference on Logic Programming. Springer-Verlag 1986.

Tick, E. and Warren, D. H. D. [1984] *Towards a Pipelined Prolog Processor.* Proc. IEEE 1984 International Symposium on Logic Programming. Also in New Generation Computing, 2, Springer-Verlag.

Turner, D. A. [1979] *A New Implementation Technique for Applicative Languages.* Software Practice and Experience, 9.

Ueda, K. [1983] *Guarded Horn Clauses.* ICOT Technical Report TR-103 (June 1983).

van Emden, M. [1982] *An interpreting algorithm for Prolog programs.* Proc. First International Logic Programming Conference, University of Marseilles. Reprinted in "Implementations of Prolog": Campbell, J. (ed.). Ellis Horwood (1984).

Van Roy, P. [1984] *A Prolog compiler for the PLM.* Master's Report Plan II, Computer Science Division, University of California, Berkeley.

Voda, P. [1986] *Choices in, and Limitations of, Logic Program.* Proc. Third International Conference on Logic Programming. Springer-Verlag 1986.

Voda, P. [1986b] *Pre-complete Negation and Universal Quantification.* Technical Report 86-9, Dept. of Computer Science, University of British Columbia.

Wadler, P. [1987] *A Critique of Abelson and Sussman or Why Calculating is Better than Scheming.* SIGPLAN Notices Vol. 22 #3 March 1987.

Walker, A. (ed.), McCord, M., Sowa, J. and Wilson, W. [1987] *Knowledge Systems and Prolog: A Logical Approach to Expert Systems and Natural Language Processing.* Addison-Wessley.

Warren, D. H. D. [1977] *Implementing Prolog - Compiling Predicate Logic Programs.* Technical Reports 39 and 40, Department of Artificial Intelligence, University of Edinburgh.

Warren, D. H. D. [1983] *An Abstract Prolog Instruction Set.* SRI Technical Note 309, Menlo Park, California.

Warren, D. H. D. [1986] *Optimizing Tail Recursion.* In "Logic Programming and its Applications": van Canemghen, M. and Warren, D.H.D (eds.). Ablex Publishing.

Wilson, W. [1985] *PureLog I: Pragmatic Logic Programming with Meta-declarations.* Ph. D. Thesis, Syracuse University.

Wise, D. [1987] *Matrix Algebra and Applicative Programming.* Conference on Functional Programming Languages and Computer Architecture, Sept. 1987.

Appendix A. Sample machine code

The code in section 2.3, "C code for deterministic append predicate" on page 21 can be transformed to machine code, for ultimate speed. These transformations cannot easily be done by a C compiler, because certain values must be kept globally in registers for ultimate performance.

A.1 Machine code for deterministic append predicate

The C code for *append* can be translated directly to machine code. Here is sample IBM/370 code.[33] The parameters are in general purpose registers 1, 2 and 3. The next free object cell's address is in register RFREE. Register RBACK points to the top of the backtrack stack (used for determining a variable's age). RZERO always has zeroes in the high three bytes, to avoid clearing it before loading a character (the IBM/370 does not clear the high bytes when loading a character). RTEMP is a temporary register. For efficiency, the object tags are 0, 4, 8, etc. instead of 0, 1, 2, etc., to avoid a shift instruction.

[33] This code is offered more as an example than as a definite method (**Warning**: the code has not been tested).

I will use a number of macros. "SWITCH reg,A1,A2,...,An" jumps to one of a list of addresses depending on a tag; it expands to something like (it is assumed that RZERO always has the high 3 bytes zero):

```
         IC   RZERO,TAG(,reg)       load tag
         L    RTEMP,SWITCH1(,RZERO) switch on type
         BR   RTEMP
SWITCH1  DC   A(A1)                 branch
         DC   A(A2)                 table
        ...                           ...
         DC   A(An)
```

"ALLOC newtag" gets a new cell from the free list; it expands to something like (the first two instructions can be eliminated if the last unallocated cell can point to an invalid address, thereby raising an addressing exception when there is no more space left):

```
         CLI  TAG(RFREE),4*TGUNALLOC next cell free?
         BNE  OVERFLOW
         MVI  TAG(RFREE),4*newtag    set tag for new cell
```

The code for the second (slower) version is:

```
START    SWITCH R1,NIL1,LISTEL1,ERROR,REF1,VAR1 switch on type of p1
NIL1 ... code omitted for brevity
REF1     L    R1,ASREF(,R1)        p1 = p1->u.asReference
         B    START                goto start
LISTEL1  SWITCH R3,NIL2,LISTEL2,ERROR,REF2,VAR2 switch on type of p3
REF2     L    R3,ASREF(,R3)        p3 = p3->u.asReference
         B    LISTEL1              back to switch
VAR2     C    RBACK,VARAGE(,R3)    is p3 newer than last
         BNH  NEWER                choice point? yes: skip
   ... code omitted: push p3's info onto reset stack
NEWER    MVI  TAG(R3),4*TGREF      p3->tag = tgReference
         ALLOC TGLISTEL            RFREE = <new> (list elem)
         LR   RTEMP,RFREE          <new> = RFREE
```

```
        L     RFREE,NEXT(,RFREE)   point to next free cell
        ST    RTEMP,ASREF(,R3)     p3->u.asReference = <new>
        ST    R1,HEAD(,RTEMP)      <new>.asListElem.head = p1
        ST    RFREE,TAIL(,RTEMP)   <new>.asListElem.tail = <new2>
NEWTAIL L     R1,TAIL(,R1)         p1 = p1->asListElem.tail
        ALLOC TGVAR                RFREE = <new2> (variable)
        ST    RBACK,VARAGE(,RFREE) <new2>.varAge = choice point age
        L     RFREE,NEXT(,RFREE)   point to next free cell
        SWITCH R1,NIL1,LISTEL1,INT1,REF1,VAR1 switch on type of p1
```

The inner loop is 19 or 23 instructions, depending on whether heap overflow is detected in-line or by an exception.

A.2 Machine code for append function

The above machine code is not optimal. As soon as a variable is found for the third argument, a variant of the loop in the first example C program can be used. The code from NEWTAIL on is replaced by:

```
inner:
    switch (p->tag) {
    case tgListElem:
        *p3 = allocListElem(p1->u.asListElem.head, &dummyCell);
        p1 = p1->u.asListElem.tail;
        p3 = &((*p3)->u.asListElem.tail);
        goto inner;
    case tgNil:
        *p3 = p2;
        break;
    case tgReference:
        p1 = p1->u.asReference;
        goto inner;
    default:
        error();
}
```

which in assembler is:

```
NEWTAIL  LA    R3,TAIL(,RTEMP)     p3 = & <new>.tail
         B     INNER2              branch to test at end of loop
INNER    ALLOC TGLISTEL            RFREE = <new> (list elem)
         ST    RFREE,0(,R3)        *p3 = <new>
         MVC   HEAD(RFREE),HEAD(R1) <new>.u.asListElem.head =
*                                    p1->u.asListElem.head
         LA    R3,TAIL(,RFREE)     p3 = &((*p3)->u.asListElem.tail)
         L     RFREE,NEXT(,RFREE)  point to next free cell
INNER2   L     R1,TAIL(,R1)        p1 = p1->u.asListElem.tail
         CLI   TAG(R1),4*TGLISTEL  while p1->tag == tgListElem
         BE    INNER
         CLI   TAG(R1),4*RGREF     or p1->tag == tgReference
         BNE   DONE
         L     R1,ASREF(,R1)       deref p
         B     INNER2                and try again
DONE     CLI   TAG(R1),4*TGNIL     if (p1->tag != tgNil)
         BNE   ERROR                 error()
         ST    R2,0(,R3)           *p3 = p2
```

This inner loop from the INNER label to the BE INNER instruction is just 8 or 10 instructions, depending on whether heap overflow is detected in-line or by an exception. On a "1 MIPS" machine, the above loop will run at over 100 KLIPS (thousands of Logical Inferences Per Second).

The inner loop has the same instruction count, for either allocating from a heap or for allocating from a stack. For reference counting, three extra instructions are needed (load, add one, store).

For this particular example, it appears as if reference counting is significantly slower than a marking garbage collector. However, a marking garbage collector must eventually scan the list and it probably will take more than three instructions to mark a cell. See section 7.12, "Reference counts and garbage collection" on page 153.

However, the Warren Abstract Machine has a very fast way of releasing storage once a query is finished – it simply pops the global heap to its initial position (some people force this using a *repeat, fail* loop). However, in the general case, where intermediate structures are created, the Warren Abstract Machine must also garbage collect its global heap.

Appendix B. Details of xpProlog syntax

XpProlog is syntactically very similar to conventional Edinburgh syntax [Periera, Warren, Byrd and Pereira 1984]. The exact syntax is not very important; to follow micro-Prolog's or Waterloo (IBM) Prolog's syntax would require very little work. I will describe xpProlog's extensions to conventional Prolog's syntax with examples.[34]

B.1 Functors and lists

Functor and lists are treated in similar ways.[35] That is, the name of a predicate may also be a variable where conventional Prolog requires that it be a name (string). XpProlog allows "pred(F(Args)) :- ..." where conventional Prolog

[34] A confession: not everything described here has been implemented – I ran out of time. The main thrust of my work was to show that pure Prolog can be implemented efficiently; I considered the exact syntax of the resulting language to be less important.

[35] I prefer the infix notation "Hd.Tl" rather than "[Hd|Tl]." C-Prolog can be persuaded to accept this by the command :- op(600, xfy, ('.')).

would require "`pred(FArgs) :- FArgs =.. (F.Args), ....`"[36] In other words, we can define *univ* by:

```
F(|Args) =.. (F.Args).   /* F(|Args) =.. [F|Args]. */
```

which also works if there are no arguments (in which case, `Args` is unified with `[]`). Also, `S() = S` but `[S] ≠ S`.

In a clause, if a goal is a functor or a list, it is called. There is no need to use *univ* before calling, nor to explicitly use the `call` meta-predicate. That is:

```
pred(F, Args) :- (F.Args), ...
```

is the same as

```
pred(F, Args) :- F(Args), ...
```

which would have to be written like this in conventional Prolog:

```
pred(F, Args) :- Call =.. F.Args, call(Call), ...
```

(Some Prologs allow leaving out the "`call`").

B.2 Lexical and syntactic details

This is a brief attempt to explain some of the intricacies of the rules for parsing Prolog. The explanation in [Clocksin and Mellish 1984] leaves a few things somewhat unexplained. Also, my method is slightly different from that used in C-Prolog [Periera, Warren, Byrd and Pereira 1983] (it is "cleaner," I think).[37]

---

[36] The parentheses around "`F.Args`" are necessary because the period could otherwise be interpreted as the end of the clause.

[37] Prolog syntax is currently being standardized by the British Standards Institute (BSI). The process is not yet finished. It will almost certainly differ somewhat from **xpProlog**.

The details of Prolog syntax are not very important; many of these details can be changed by simply changing some lexical tables.

First, the basic lexical items:

string Also called "name," "atom" or sometimes "id." This is a lower-case letter followed by any number of letters and digits. Any other characters can be included if the entire item is surrounded by single quotes ("'"). Unprintable characters and single quotes can be included if they are preceded by a backslash ("\"), following C syntax (for example, `'\\\'a\'\n'` which, when output, produces a single backslash, a beep, a quote and a new-line).

variable name An upper-case letter or underscore followed by any number of letters, digits or underscores. A single underscore is an anonymous variable, unique from all other variable names in the clause.

quoted string A string surrounded by double quotes ("""). Again, C syntax conventions are followed for unprintable characters and double quotes. "`Abc`" is equivalent to `['A','b','c']`.

number A sequence of digits, optionally preceded by a negative sign ("-") and optionally containing a decimal point.

However, there will be little difficulty in modifying **xpProlog** to conform to the new standard.

delimiters Most other characters are considered as delimiters ("!@#$%¬&*()-" etc.). Except for the bracketing symbols ("()[]{}"), delimiters are treated like strings. When an operator is defined (by the op built-in predicate) and the operator's name is made up entirely of delimiters, then the entire string is treated as a single item (for example, "-->" or ":-").

white space Ignored between lexical items – includes blanks, tabs, new lines and comments. Comments may either be bracketed by "/* ... */" or marked by "%" which causes everything up to the end of line to be ignored.

statement terminator A period ("."). Unfortunately, it can also be used as a decimal point in numbers and as the "cons" operator for lists. Therefore, "p(X):-X=a.b." must be written "p(X):-X=(a.b)." to avoid ambiguity (the former phrase would be interpreted as two clauses: "p(X):-X=a." and "b.").

An operator is defined by the op built-in predicate:

```
:- op(Priority, Associativity, OpName).
```

The operator name ("OpName") must be enclosed in single quotes if it contains any delimiter characters. The operator name is then added to a lexical table so that it will be subsequently treated as a single item.

Unlike "normal" compilers, higher priorities binds looser.[38] Here are the usual arithmetic operators:

```
:- op( 500, yfx, +).
:- op( 500, yfx, -).
:- op( 500,  fx, +).        % unary (prefix)
:- op( 500,  fx, -).        % unary (prefix)
:- op( 400, yfx, *).
:- op( 400, yfx, /).
:- op( 400, yfx, /).
```

The associativity is of the forms:[39]

`xfx`	`xfy`	`yfx`	`yfy`	(for infix operators)
`fx`	`fy`			(for prefix operators)
`xf`		`yf`		(for postfix operators)

If there are no parentheses, a `y` means that the argument can contain operators of the same or lower precedence (same or tighter binding) while `x` means that the argument can contain operators of strictly lower precedence (tighter binding). Thus, `yfx` is left-to-right and `xfy` is right to left. Some operators (such as comparisons) are defined `xfx`. This prohibits, for example, `a < b < c` although `(a < b) < c` would be legitimate (but meaningless).

[38] The BSI committee is proposing to change this.

[39] In IBM (or Waterloo) Prolog, the associativity is one of `prefix`, `suffix`, `lr` or `rl` which doesn't allow quite the same degree of control as does C-Prolog's values. On the other hand, IBM Prolog is more readable.

Some ambiguities are still possible. An `xfy` operator next to a `yfx` operator can be parsed in two ways if they have the same priority. Such cases are flagged as errors by the parser ("ambiguous operator juxtaposition").

Note that monadic (prefix and postfix) operators also have priorities. For example, `?- X is 1+2.` is parsed `?- (X is (1+2)).` whereas `-a+b` is parsed `(-a)+b.`

An operator is treated as such only if it is in a position which allows it to be an operator.[40] For example, in "`a opx + b`," the `opx` can potentially be either a postfix operator or an infix operator. If `opx` is a prefix operator, then the "+" must be an infix operator ("`(a opx) + b`"); if `opx` is an infix operator, then the "+" must be a prefix operator ("`a opx (+b)`"). The decision is done left-to-right with no backtracking.

C-Prolog distinguishes between operators and functor names. It considers "`not x`" and "`not(x)`" to be distinct. **XpProlog** does not make this distinction. Furthermore, there is an ambiguity with prefix operators and comma operators (recall that `?-op(1000,xfy,',')` – that is, comma is a right-to-left infix operator). If "+" is a prefix operator, then `+(1,2)` can be parsed as either the binary operator + applied to the two operands `1` and `2` or it could be the unary operator applied to the single operand which is itself a binary operand applied to the two operands `1` and `2`. **XpProlog** takes the former meaning; if the latter is desired, it should be written "`+((1,2))`." (C-Prolog distinguishes between the

40 Some Prologs, including the draft BSI standard insists that an operator cannot be enclosed within quotes. I have not implemented this, although it could easily be done.

cases by looking for a blank after the "+"; I (and the BSI standard) consider this to be a kludge).

Note that operators may be enclosed in parentheses to have them considered as ordinary strings. For example, `(+)*(:-) = (*)(+, :-)`.[41]

Lists may be entered either in bracket form or dotted form. Here are some examples (IBM and Waterloo Prologs use "!" instead of "|" and also allow curly brackets ("{}") instead of square brackets ("[]")):

```
[a | b] = a . b
[a]     = [a | []]      = a . []
[a, b]  = [a, b | []]   = a . b . [] = a . ( b . [])
```

Curly brackets are also allowed, for grammar rule notation. Note that here comma is used as an operator rather than a separator.

```
{a}     = '{}'(a)
{a, b}  = '{}'((a,b))    = '{}'(','(a,b))
```

B.3 Critique of Prolog's syntax

Prolog's syntax badly overloads several characters, particularly

"." (period):

- end of clause
- "cons" for lists

41 With the BSI convention, `'*'('+', ':-')` achieves the same affect.

- decimal point in a number

"," (comma):

- argument separator inside predicates, functors and lists
- and-operator, separating the goals of a predicate
- ordinary operator, for example inside "{}"

Period is especially troublesome. For example, "`1.2.3.`" can be interpreted as "`[1,2,|3]`" or "`[1|2.3]`" or "`[1.2|3].`"

Some other nasty surprises are possible because most "syntactic sugar" is done using monadic and diadic operators. Many programmers have been unpleasantly surprised by what happens with "`p->q,r;s,t`" (the "... `->` ... ; ..." notation is used for *if-then-else*).

I propose the following solution:

"."

- Use only as a decimal point
- "cons" is represented by the colon ":" (I consider the "`[A|B]`" to be hard to read).
- end of clause can normally be inferred (as in BCPL [Richards and Whitby-Stevens 1979]); in the few places where it cannot be inferred, an explicit "`end`" is used (alternatively, ";" could be used; "or" probably should use the more obvious "|" symbol instead).

“,”

- Used only as an argument separator.
- “And” (“&”) is used as a goal separator. This “&” can also be an operator.
- “`{a,b}`” is interpreted as “`'{}'(a,b)`” – this can easily be handled in a processor for grammar rules because “`'{}'(|Args)`” will put all the arguments into “`Args`.”

Prolog's method of entering predicates by listing out the clauses makes things a little difficult for a compiler, because the compiler cannot know when a predicate has been completely defined. Normally, all the clauses for a predicate are kept together; in fact, separated clauses usually indicate an error in typing the predicate. Therefore, **xpProlog** enforces that all the clauses for a predicate appear together.

B.4 Debugging extensions

Misspelling a variable name or predicate name is a very common error. **XpProlog** gives an warning message if a variable name is used only once within a clause (this message can be suppressed by prefixing the name by an underscore).

At run-time, if an undefined predicate is tried, an error message is output and the user is prompted for a definition of the predicate (“query the user”).[42] This

42 This has not yet been implemented; currently the user is prompted only for a simple success or failure. For a full discussion of “query the user,” see [Sergot 1983].

can be suppressed by defining the predicate as always failing (using the built-in predicate "`fail`").

One other troublesome point in Prolog is what happens if a predicate is tried with the wrong type of argument. Most such cases can be caught if the programmer indicates that a predicate should always succeed. If the programmer writes "`?-neverfail(pred(_,_)).`" then when "`pred/2`" fails, an error message is produced. This is done by automatically adding an extra catch-all clause to "`pred/2`" which produces the error message.

Type inferencing can help in detecting undefined predicates and wrong types to predicates. It is discussed in section 5.10, "Speeding up deterministic predicates – modes and types" on page 112.

Appendix C. Implementation status

An interpreter for the xpPAM has been completed. It includes a simple assembler and loader.

The implementation is incomplete in the following ways:[43]

- Some opcodes are implemented inefficiently. The opcodes should be combined with the operand types to allow faster decoding (as mentioned in section 3.7, "Machine instruction format" on page 36). This would increase the number of cases in the interpreter by a factor of three or four, while approximately doubling its speed.

- The implementation of delays is inefficient. Currently, a variable which caused a delay has a flag set; when such a variable becomes instantiated, it causes a search for a predicate which can be resumed. A much more efficient way would be to have "uninstantiated variable which caused a delay" to get a separate tag and to have the cell point at a list of predicates which depend on it (as described in section 4.4, "Delays" on page 75).

[43] I simply ran out of time and concentrated on writing this report; the remaining work will probably take a few months to finish.

- Some opcodes have not been implemented:

  - `testskip`
  - `gete`
  - `caseGoto`
  - `cutAt`
  - `chopBack`
  - `delayCost`
  - `mkTHunk`
  - `delayRec`

- Only some built-ins have been implemented. A complete list of the existing built-ins is given in Appendix D, "Built-in predicates" on page 300 together with their implementation, to indicate how more built-ins can be easily added.

- The "virtual choice depth" is not recorded on the backtrack stack (it can easily be added).

- The implementation uses three stacks. This could be changed to two by combining either the execution and backtrack stacks (the WAM choice) or the reset and backtrack stacks (more preference). Note that combining the reset and backtrack stacks requires saving the "virtual choice depth."

- User defined unification and complex indeterminates have not been implemented.

- The unification algorithm is recursive, not using the Deutsch-Schorr-Waite algorithm (recursive unification is detected as described in section 3.10, "Unification" on page 40). However, freeing memory does use the D-S-W algorithm.

- An optimizing compiler exists only for an earlier version of the abstract machine. Some re-writing is necessary for the present abstract machine.

- An un-optimizing compiler has been written. It generates `xpPAM` assembler. It has not yet been integrated (bootstrapped) with the interpreter. Consequently, predicates must be first run through the compiler, then processed by the interpreter. A fully integrated compiler will implement the compile on first use strategy described in the thesis: a predicate's text is compiled when the first attempt is made to execute it and the compiled code replaces the text.

- The compilers only produce abstract machine code; they do not compile directly to machine code (as in the chapter 2.0, "Fast append on a conventional machine" on page 15). It is not clear how translation to actual machine code should be done: by using abstract machine instructions as templates and then doing "peephole" optimizations to recognize special cases; or by compiling directly to machine code, using the abstract machine instructions as a guide to the algorithm.

- The compilers do not handle "?" notation nor *proceed* declarations; they must be added by hand to the generated assembler.

- The "closed predicate" notion has not been implemented, nor has negation transformation (section 10.1, "Negation" on page 207).

- The parser contains code to properly handle Prolog style operators (albeit with slightly different rules than conventional Prolog – see Appendix B, "Details of xpProlog syntax" on page 285.). However, the compilers do not yet handle the *if-then-else* constructs.

- The array and I/O predicates described in section 11.0, "Arrays and I/O done logically and efficiently" on page 222 have not been implemented.

- The compilers do not handle explicit *if-then-else*. The optimizing compiler can detect *if-then-else* situations and generate appropriate code. The compilers also do not handle the special *else* predicate nor the *suchthat* notation.

- Meta-variables have not been implemented.

- Object paged virtual memory has not been implemented; the compacting scheme in section 3.15, "Virtual memory" on page 61 has been implemented.

To test the speed of the final interpreter, a number of test programs have been written to test the speed of deterministic append (the inner loop of the naïve reverse benchmark). These try the optimized abstract machine (with opcodes and operand types combined) and also directly generated machine code (effectively, removing the interpreter loop and directly executing the code for the various abstract instructions). These have given results in the speed range which has been advertised for some of the better commercial implementations (about 20KLIPS on a 1 MIPS machine), without any attempts to optimize for the particular machine. In addition, the reference counting mechanism has been experimentally turned off, giving approximately 15% speed-up, as suggested by the Smalltalk papers (see section 7.12, "Reference counts and garbage collection" on page 153).

Appendix D. Built-in predicates

Here is a list of the built-in predicates which have been implemented. The final list will be much larger. These built-ins can be considered as extensions to the basic xpPAM. The following gives definitions of the built-ins, plus their xpPAM code.

Some of these built-ins are non-logical. They exist for compatibility with conventional Prolog, and also as building blocks for logical predicates.

call(X) X must be instantiated to a term which can be interpreted as a goal. The call(X) goal succeeds by attempting to satisfy X. X may be an atom, a functor or a list.

```
code(call, 1, [], [    % call(X) :- X.
    lCall(0.f)]).
```

X , Y The same as call(X), call(Y).

```
code(',', 2, [], [    % (X, Y) :- call(X), call(Y).
        link,
        push(1.f),
        call(0.f),
        pop(0),
        unlink,
        lCall(0.f)]).
```

X & Y The same as `call(X), call(Y)`. The definition is identical to "," (above).

X ; Y Can be defined by

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

```
code(';', 2, [], [     % (X ; Y) :- X.   (X ; Y) :- Y.
        pushB(1.f),
        mkCh(c12),
        lCall(0.f),
    label(c12),
        popB(0),
        lCall(0.f)]).
```

'<goal>' This is an internal predicate which is used to handle top-level queries. It is used as template which is modified.

The top level of the interpreter produces two structures: one with all the variable names replaced by uninstantiated variables and the other showing the bindings. For example:
`foo(A,B,A), bar(A).`
generates something like this (note that "," is the "and" predicate given above); the *more* and *nomore* built-ins are described later):
`(foo(_5,_7,_5), bar(_5)), '<more>'(['A'._5, 'B'._7]).`

This is compiled to:

```
/* constant 0: top level goal to call:
               foo(A,B,A), bar(A). */
/* constant 1: list of variable names (for <more>):
               ['A'._5, 'B'._7] */
  mkChAt  1,fin    % must have a choice point on the stack
  eq      n0, c0   % get goal into reg 0
  link
  push    f1       % remember choice point
  call    f0       % call the goal
  pop     n1       % get choice point into reg 1
  unlink
  eq      n0, c1   % arg for <more>
  builtin "more"
  free    f0
  free    f1
  stop
fin:
  builtin "noMore" % always succeeds
  stop
```

X = Y unifies X and Y.

```
code((=), 2, [], [          % X = X.
        eq(0.f, 1.f),
        return]).
```

true always succeeds.

```
code(true, 0, [], [         % true.
        return]).
```

fail always fails

```
code(fail, 0, [], [         % fail :- <fail>.
        fail]).
```

X is Y Y must be instantiated to a structure that can be interpreted as an arithmetic expression. X is then unified with the result. All arithmetic is done double precision floating point (note that this gives perfectly accurate results for integers up to a few billion). I am investigating further enhancements to *is*. For example, we may add more constants (by adding to the associative memory) or allow calling user-defined predicates. *Is* delays if anything in the structure is uninstantiated.

```
code(is, 2, [], [                % Left is Right
        builtin(4),              % eval: r2 := eval(r1)
        free(1),
        eq(0.f,2.f),
        return]).
```

The "eval" built-in opcodes expects register 1 to contain a structure to be evaluated; the result is put into register 2 which must be empty (on failure, register 2 is left alone). If there is an uninstantiated variable to be evaluated, the built-in delays (this is slightly inefficient if the expression is complex; the method of defining *is* as given in section 6.5, "Equality: "is" and "="" on page 133 would be preferable).

Here are some further examples of using the "eval" built-in opcode. They implement *plus* which works if any argument is uninstantiated (delaying if two or more are uninstantiated and doing addition or subtraction otherwise); and *less-than*:

```
code(plus, 3,                       % r0+r1 = r2
/*0*/  [const([]),
/*1*/   const('+'),
/*2*/   const('-')],
       [eq(3.n,2.f),                % move r2 into r3,
    label(restart),                 %      freeing r2
        varGoto(0, var0),
        varGoto(1, var1),
    % from here, only r3 is possibly uninstantiated
        eqlst(4.n, 1.f, 0.c),       % r4 = r1 . []
        eqlst(5.n, 0.f, 4.f),       % r5 = r0 . r4
                                    %    = r0 . r1 . []
        eqlst(1.ns, 1.c, 5.f),      % r1 = '+' . r5
                                    %    = '+'(r0, r1)
        builtin(4),                 % eval: r2 := eval(r1)
        free(1),
        eq(2.f, 3.f),
        return,

    label(var0),
        varGoto(1, delay),
        varGoto(3, delay),
    % from here, only r0 is possibly uninstantiated
        eqlst(4.n, 1.f, 0.c),       % r4 = r1 . []
    label(v1),
        eqlst(5.n, 3.f, 4.f),       % r5 = r3 . r4
                                    %    = r3 . r1 . []
        eqlst(1.ns, 2.c, 5.f),      % r1 = '-' . r5
                                    %    = '-'(r3,r1)
        builtin(4),                 % eval: r2 := eval(r1)
        free(1),
        eq(2.f, 0.f),
        return,

    label(var1),
        varGoto(3, delay),
    % from here, only r1 is possibly uninstantiated
        eqlst(4.n, 0.f, 0.c),       % r4 = r0 . []
        eq(0.n, 1.f),
        goto(v1),
    label(delay),
        nonvarGoto(0, delay0),
        delay(0, restart),
    label(delay0),
        nonvarGoto(1,delay1),
        delay(1,restart),
    label(delay1),
        nonvarGoto(3, delay3),
        delay(3, restart),
    label(delay3),
        builtin(3)]).           % error/3
                                %  (shouldn't happen)

code((<), 2,                    % A < B :- 1 is (A < B)
       [const([]), const(1), const(<)],
       [eqlst(3.n, 1.f, 0.c),   % r3 = B . []
        eqlst(2.n, 0.f, 3.f),   % r2 = A . r3
                                %    = A . B . []
        eqlst(1.ns, 2.c, 2.f),  % r1 = '<' . r2
                                %    = '<'(A,B)
        builtin(4),             % eval: r2 := eval(r1)
        free(1),
        eq(2.f, 1.c),           % r2 = 1 ?
        return]).
```

X \= Y Succeeds if X and Y cannot be unified. It delays if either argument is insufficiently instantiated.

```
code((\=), 2, [], [         % noteq: r0 \= r1
        builtin(5),         % noteq
        free(0), free(1),
        return]).
```

This is an old form which should be replaced by:

```
code((\=), 2, [], [
        testskip(0.f, 1.f), % delays if necessary
        fail                % success becomes failure
        return]).           % failure becomes success
```

Note that *not equal* must be inside a predicate. The *testskip* (or *builtin(5)*) will cause its enclosing predicate to delay, which gives the appearance of replacing the opcode by a *return*. Thus, if the *not equal* were in-line, it would cause execution to not consider any of the following goals until the *not equal* had been resumed.

name(A,L) The characters for the atom A are the list L. For example, `name(ab,[a,b])` or `name(ab,"ab")`. Note that this is different from C-Prolog which puts the ASCII codes into the list but **`xpProlog`** puts the individual characters in.

```
code(name, 2, [], [                 % name(String, List)
        varGoto(0, var0),
        builtin(6),                 % r2 := strToList(r1)
        free(0),
        eq(1.f,2.f),
        return,
    label(var0),
        varGoto(1, var01),
        builtin(7),                 % r2 := listToStr(r0)
        free(1),
        eq(0.f,2.f),
        return,
    label(var01),
        delayOr(0,0),
        delay(1,0)]).
```

The built-in opcode “strToList” turns a string into a list. It expects a string in register 0 and assigns the list expansion of the string to register 2.

The built-in opcode “listToStr” turns a list into a string. It expects a list in register 1 and assigns the string value to register 2.

include F switches input to be the file F. This will probably change to the C-Prolog “consult.”

audit audits the memory to look for wrong reference counts, etc. (This is not a true predicate right now.)

code(N,NP,C,M) defines the name N with NP parameters by the code defined by the constant list C and the machine code list M. The format is given in an appendix. The old definition, if any, is replaced. Note that the code segment constants always have one level of indirection so that replacing a predicate automatically changes it everywhere it is used. If a predicate is used which hasn't yet been defined, the predicate is automatically defined to produce a warning message (“Undefined predicate”) and then fail. A “query the user” facility can be added by simply replacing the default “undefined” predicate.

```
code(code, 4, [], [          % code(Name, Arity, ConstList,
                                       %      MachCodeList)
        builtin(8),          % code(r0,r1,r2,r3)
        free(0), free(1), free(2), free(3),
        return]).
```

delcode(N,NP) removes the definition of the name N with NP parameters. Again, because code segments are always handled with one level of indirection, this removes the definition everywhere (it is actually replaced by the "Undefined predicate" warning).

```
code(delcode, 2, [], [          % delcode(Name, Arity)
        builtin(9),             % delcode(r0,r1)
        free(0), free(1),
        return]).
```

dbgFlag(X) turns off debugging flag "X" ("`-dX`" in the command line can set this on initially). "`dbgFlag(' ')`" turns on the general debug flag. About the only other useful thing is `dbgFlag(t)` and sometimes `dbgFlag(i)` to do tracing.

```
code(dbgFlag, 1, [], [          % dbgFlag(Char)
        builtin(10),            % dbgFlag(r0)
        free(0),
        return]).
```

nodbgFlag(X) turns on debugging flag "X."

```
code(nodbgFlag, 1, [], [        % nodbgFlag(Char)
        builtin(11),            % nodbgFlag(r0)
        free(0),
        return]).
```

op(P,A,O) defines operator O with priority P and associativity A (see section B.2, "Lexical and syntactic details" on page 286 for parsing and the standard operators).

```
code(op, 3, [], [            % op(Prio,Assoc,Name)
        builtin(12),         % op(r0,r1,r2)
        free(0), free(1), free(2),
        return]).
```

tableAssign(T,I,V) assigns value V to the associative table T at index I. V can be anything at all (it should be instantiated or else strange things will happen); T and I must be strings (atoms).

```
code(tableAssign, 3, [], [   % assign(Table,Item,Val)
        builtin(13),         % assign(r0,r1,r2)
        free(0), free(1), free(2),
        return]).
```

tableFind(T,I,V) finds value V in the associative table T at index I. T and I must be strings (atoms).

```
code(tableFind, 3, [], [     % tableFind(Table,Item,Val)
        builtin(14),         % eval: r3 := find(r0,r1)
        free(0), free(1),
        eq(2.f,3.f),
        return]).
```

tableDelete(T,I) deletes the associative table T at index I. T and I must be strings (atoms).

```
code(tableDelete, 3, [], [   % tableDelete(Table,Item)
        builtin(15),         % delete(r0,r1)
        free(0), free(1),
        return]).
```

Incidentally, a backtracking version of *tableAssign* would be:

```
tableAssignB(T,I,V) :-
    if exists VOld suchthat tableFind(T,I,VOld) then
        (tableAssign(T,I,V); tableAssign(T,I,VOld))
    else
        (tableAssign(T,I,V); tableDelete(T,I))
    endif.
```

The semi-colon (";") is an *or*. The first time in, the `tableAssign(T,I,V)` is executed. On backtracking, the `tableAssign(T,I,Vold)` or `tableDelete(T,I)` is executed to undo the affect of the original `tableAssign`.

put(Item) output one item on the output "stream." No quote marks are added (`display` adds quote marks).

```
code(put, 1, [], [              % put(Item)
        builtin(16),            % put(r0)
        free(0),
        return]).
```

nl output a new line.

```
code(nl, 0, [const('\n'), call(put,1)], [   % nl()
        eq(0.n,0.c),
        lCall(1.c)]).           % put('\n')
```

display(Item) display an item, in such a way that it can be read back in (that is, with quote marks added as necessary).

```
code(display, 1, [], [          % display(Item)
        builtin(17),            % display(r0)
        free(0),
        return]).
```

D.1 Built-in opcodes.

A built-in opcode is not normally used directly; rather it is embedded inside a "shell" predicate which sets up the registers for it. After each built-in is given the shell predicate (often with the same name) which set it up.

The built-in opcode is used to extend the instruction set of the basic inference machine. A built-in may succeed, fail or delay, not necessarily in the same way as its "shell" predicate.

Most of the built-in opcodes have been given above.

Undef: an undefined predicate gets set to the following code:

```
built-in "undef"
fail                % always does this
```

The *undef* builtin uses information from the current code segment to display information about the predicate. The built-in always succeeds and frees all the registers that were set up for the call (again, by using information from the current code segment). The instruction following the built-in is a *fail*, so the predicate fails (the built-in was made to succeed so that it could also be used for a query-the-user facility).

More is used with `'<goal>'` to prompt the user for more solutions. *More* expects register 0 to contain a list of variables (in the form `[name1.value1,name2.value2 , ...]`) and register 1 to contain the original choice point. *More* displays the values.

- If the list of variables is empty, it succeeds.
- If the backtrack stack is empty (no other solutions possible), it succeeds.
- If input is from the terminal, it waits for the user to type in either a semi-colon (";") or just *<return>*. In the former case, `'<more>'` fails to cause backtracking; in the latter case, it succeeds so that no more solutions are produced. The initial choice point must have been created by `'<goal>'` so that a final backtrack has somewhere to go to.
- If input is from an included file, it either succeeds of fails depending on the value of a user-settable flag — if the flag is on, backtracking will generate all possible answers.

NoMore takes no registers. It displays "No (more) solutions." and succeeds. It should always be followed by a *stop* instruction.

Error uses information from the current code segment to display information about the predicate. This built-in is used directly in user code; typically for a case from a *swXVNL* instruction for an invalid argument. This built-in always fails.

The end

This page intentionally left blank, to indicate the end of the thesis.