The Berkeley PLM Instruction Set: An Instruction Set for Prolog

Barry Fagin
Tep Dobry
Division of Computer Science
University of California
Berkeley, CA 94720

ABSTRACT

This document describes the instruction set of the Berkeley Programmed Logic Machine (hereafter referred to as the PLM). The PLM instruction set is heavily influenced by the Prolog instruction set of David Warren, [1], with some minor modifications involving cut and cdr-coded data structure representation [2].

The report is divided into two parts. The first part describes the abstract resources of the PLM and the data types it manipulates, in order to facilitate an understanding of how the instruction set works. The second part describes the instruction set itself.

TABLE OF CONTENTS

CHAPTER 1

The Data Types and Resources of the PLM

This chapter will first explain the basic data types that the PLM manipulates, and show examples of each. The resources of the abstract machine will also be outlined. The PLM uses four different areas of memory: the heap, the stack, the trail, and the push-down list (PDL). It also has several special registers, designated by the symbols An, B, CP, E, H, HB, N, P, PDL, S, and TR. The uses of all these resources will be explained.

1. Data Types

The PLM manipulates four kinds of data types: structures, lists, variables, and constants. The type of a data word is indicated by an appropriate tag.

1.1. Constants

Constants can be of several types, including integers, atoms, and floating point values. Small integers are stored directly in the data word itself, while atoms and floating point values contain pointers to the appropriate item in the code space.

1.2. Variables

A variable is simply a data word with the tag 'variable' in the most significant byte, whose contents are an address of some other data word. As an optimization, the PLM uses a bit to indicate whether or not the variable is bound. (By contrast, Warren's machine represents unbound variables as pointers to themselves).

1.3. Lists

Lists are represented by a word with the list tag, pointing to the first entry of the list. List entries are one word long, and use cdr-coding to improve memory efficiency. Conventional list representation uses two words for each entry: the car, which contains the list entry itself, and the cdr, which points to the remainder of the list. With cdr-coding, if the cdr cell corresponding to a list entry represented conventionally would point to the next word in memory, then that cell is omitted. When this is not the case, the cdr cell is left in memory, with a bit set to indicate that the word is an explicit cdr cell. This bit is called the cdr bit. Thus, to determine the location of the next entry of a list, one simply examines the next contiguous word in memory. If the cdr bit is off, then that cell is the next entry. If it is on, then the location of the next entry is pointed to by the contents of the cell.

Lists always end with a cell whose cdr bit is set.

1.4. Structures

Structures are simply lists with principal functors. They are represented by a word with the structure tag, whose contents are a pointer to the principal functor of the structure, followed by the arguments of the structure.

2. The Resources of the PLM

The important resources of the PLM deal are the memory and registers. This section explains the purpose of each of the special registers of the PLM and how memory is partitioned.

2.1. Register Usage

The current state of a Prolog computation is defined by certain registers containing pointers into the data memory. This memory is divided into four areas: the heap, the stack, the trail, and the PDL. The purpose of each will be explained in more detail after the PLM registers are discussed.

The PLM makes use of the following special registers:

PLM Register Set

A1 - An:	the Argument registers (also referred to as X1-Xn)
P:	the Program counter
CP:	the Continuation Pointer
E:	the Environment pointer
B:	the Backtrack pointer
TR:	the Trail pointer
H:	the Heap pointer
HB:	the Heap Backtrack pointer
S:	the Structure pointer

The purpose of each of these registers is explained below.

A1 - An

The Argument registers: contain the arguments of a Prolog goal. For example, for the PLM to execute the Prolog query "d(4,5,6)?", registers A1, A2, and A3 would be loaded with the tagged words representing the constants 4, 5, and 6 respectively, and then the code for the procedure d would be entered. In the PLM, n = 8.

Occasionally, these registers are referred to as X1 - Xn. This is done whenever the register is used by the compiler as a temporary location, instead of an actual argument to a goal. However, Xn and An refer to the same register.

P The Program pointer: contains the address of the next instruction to execute.

CP The Continuation Pointer: contains the address of the next instruction to execute should the current goal succeed. For example, when the PLM begins to execute the code for procedure "h" in the clause "f(X) :- g(X), h(X), i(X)", the CP would contain the address of the code corresponding to the call to i. In other words, the CP functions like a return pointer for a subroutine call.

E The Environment pointer: contains the address of the last "environment" pushed on the stack. (Environments will be explained shortly).

B The Backtrack pointer: contains the address of the last "choice point" pushed on the stack. (Choice points will also be explained shortly).

TR The Trail pointer: points to the top of the trail.

H The Heap pointer: points to the top of the heap.

HB The Heap Backtrack Pointer: the value of H at the time the last choice point was placed on the stack.

S The Structure Pointer: Used to address elements of structures and lists on the heap. Points to the current element of a structure or list being addressed.

2.2. Data Memory Allocation

The data memory of the PLM is partitioned into three stacks: the control stack, the heap, and the trail. In addition, a scratchpad area, the push-down list, (PDL) is provided.

2.2.1. The Control Stack

The control stack (hereafter called "the stack") is the area in memory used for storing control information. Two kinds of objects may appear on the stack: environments, and choice points. Both of these objects are placed on the stack by special PLM instructions, as we shall see shortly.

2.2.1.1. Environments An environment corresponds to the saved state of a prolog clause: it contains pertinent register values, and what are known as "permanent" variables. Permanent variables are variables needed by more than one goal in the body of a clause; they must be saved so that succeeding goals can access them.

For example, consider the following Prolog clause:

f(X,Y) :- g(X), h(X,Z).

At the beginning of executing the code corresponding to this clause, an environment will be allocated for it on the stack, and the data word representing the variable X would will be stored within it. Thus, after executing the code for the procedure g, h will be able to access X by referring to location of X on the stack. X is a "permanent" variable because it occurs more than once in a clause, and its last occurrence is after the first goal. If its value were not saved in an environment on the stack, other goals would not be able to reference it.

By contrast, the clause

f(X,Y) :- g(X), h(Z).

has no permanent variables, because the second occurrence of the variable X is in the first goal. Since X will already be present in the first argument register when the code for f is executed, (why this is so will be shown in some examples), its location does not have to be saved. The clause

f(X,Y) :- g(Z), h(W).

similarly has no permanent variables, because no goal requires access to the variables of another. Formally, a variable is temporary if it occurs in at most one goal of a clause, where the head is considered part of the first goal. All variables that are not temporary are permanent.

Environments also contain the values of certain registers, to enable restoration of the state of a computation when the last goal in the clause succeeds. Environments contain the following register values:

CP : where to continue once this clause succeeds
E : location of last environment on stack
N : size of last environment
B : location of last choice point (needed for implementing cut)

2.2.1.2. Choice Points

A choice point is a group of data words containing sufficient information to restore the state of a computation if a goal fails, and to indicate the next procedure to try. Choice points are placed on the stack by special PLM instructions when a procedure is entered that contains more than one clause that can unify with the current goal. For example, as the PLM executes the following Prolog program fragment

```
g(X) :- f(X), h(X, X).
g(X) :- a(X), b(X, Y).

g(X)?
```

a choice point would be placed on the stack when the first clause is entered, because should it fail an alternative clause exists which is to be tried as well.

Choice points contain the following register values:

An: the contents of the argument registers
E : location of last environment
CP : address of next clause to execute should this one succeed
B : location of previous choice point
TR: the value of the trail pointer when choice point built
H: the top of the heap when choice point built
N: the number of permanent variables in the current environment
L: address of next clause to try should current goal fail.

2.2.2. The Heap

The heap is the area of data memory used for the storage of lists and structures, which are too cumbersome to be kept in environments on the control stack. While the heap is also used for "globalizing" unsafe variables (this will be explained later), its primary purpose is the storing of lists and structures. The choice of name for this area of memory is unfortunate, because it is actually allocated like a stack, with successive entries pushehd onto it, and deallocated in blocks when backtracking occurs.

2.2.3. The Trail

When a variable becomes bound during the course of a Prolog program, it may become necessary to undo the binding when backtracking is done. Thus some method is needed for keeping track of all bindings that are to be undone when the current goal fails, so that the variables they refer to can be unbound again. For example, in the Prolog program

f(a).
f(b).
g(b).

f(X), g(X)?

X would first be bound to a, but "g(a)" would have no solution. Thus the binding of X to a must be undone. X will then unify with b, and "g(b)" will succeed.

The PLM uses a small stack called the Trail to handle the necessary bookkeeping for bindings that will have to be undone upon goal failure. This stack is addressed by the TR register. When a binding is trailed, a pointer to the variable just bound is pushed onto the Trail, (in the previous example a pointer to the variable X), and the TR register incremented. Upon goal failure, all variables pointed at by pointers on the trail, from the top of the trail down to the previousy saved TR value in the current choice point, are reset to unbound variables. This is done as part of the 'fail' operation, explained in the section on fundamental operations.

It should be noted that not all bindings need to be trailed. Suppose the variable being bound is on the stack. If it is located above the current choice point (assuming the stack grows upwards) then it will be thrown away on goal failure; hence the binding will not have to be explicitly undone. Similarly, if the variable being bound is on the heap, then if it is located above the address in the HB register (assuming the heap grows upwards) then it will be discarded on goal failure. Thus whenever a binding is made, we compare its address with the appropriate register (B or HB). Only if the address is less than the B register (for a variable located on the stack) or the H register (for variables located on the heap) is the address of the variable pushed onto the trail.

2.2.4. The PDL

The PDL is a small stack used for the unification of nested structures and nested lists. Consider the problem of unifying the lists '[a,[b,c,d],e]' and '[a,[b,c,d],f]'. Both objects are lists, and their first elements match. The second element in each object is also a list, and each of their elements match. However, the last element of each list is inaccessible. It is pointed to by the cdr cell after the sublist '[b,c,d]', whose address we neglected to save. This problem is solved by pushing pointers to points where unification of a nested data object is to continue onto a stack; in the PLM, this stack is the PDL. When the end of a substructure is encountered, the topmost entry on the PDL is popped off and unification coninues at the point that entry indicates.

Either depth first or breadth first traversal of nested structures is possible. However, since Prolog structures tend to be long rather than deep, depth first traversal uses less PDL space and is hence preferable. With depth first traversal, the maximum value of the PDL will be the maximum depth of nesting of a structure in the program, whereas with breadth first it will be the maximum number of arguments of a structure or entries in a list. The former tends to be much smaller than the latter.

CHAPTER 2

The Instruction Set of the PLM

The PLM executes a modified version of the Prolog instruction set defined by David Warren. The instructions can be divided into six basic groups. In addition, there are some basic operations performed by the instructions that must be supported. This chapter explains the uses of each instruction and the basic operations.

1. The Instruction Groups

The instruction set of the PLM is made up of six subsets: indexing instructions, procedure control instructions, clause control instructions, get instructions, put instructions, and unify instructions, grouped as follows:

PLM Instruction Set

INDEXING INSTRUCTIONS
- switch_on_term
- switch_on_constant
- switch_on_structure

PROCEDURE CONTROL INSTRUCTIONS
- try
- retry
- trust
- try_me_else
- retry_me_else
- trust_me_else
- fail
- cut
- cutd

CLAUSE CONTROL INSTRUCTIONS
- proceed
- execute
- call
- escape
- allocate
- deallocate

GET INSTRUCTIONS
- get_variable
- get_value
- get_constant
- get_nil
- get_structure
- get_list

PUT INSTRUCTIONS
- put_variable
- put_value
- put_unsafe_value
- put_constant
- put_nil
- put_structure
- put_list

UNIFY INSTRUCTIONS
- unify_void
- unify_value
- unify_variable
- unify_constant
- unify_cdr
- unify_nil

Most of these instructions take arguments. These arguments are either constants, register names, or labels representing compiled addresses.

2. How Procedures are Compiled: a Preview

All compiled Prolog procedures have the same basic format: [3]; they consist of a procedure block followed by one or more clause blocks. A procedure block has the following format:

```
switch_on_term     L_const, L_list, L_struct
L_var:
        try            Clausex
      retry            Clausey
        ...
      trust            Clausen
L_const:
        switch_on_constant   size of hash table
      hash table
      ...
L_list:
      try              Clausez
      ...
      trust            Clausem
L_struct:
      switch_on_structure  size of hash table
      hash table
      ...
```

The procedure block contains indexing and procedure control instructions to determine which clauses will be executed, based on the type of the data item in the first argument register.

A clause block consists of the compiled code for a Prolog clause. A clause with subgoals SG1, SG2, ... SGn would compile into the following clause block:

```
Clause:  allocate
         get instructions
         put instructions
         call SG1
         put instructions
         call SG2
         ...
         deallocate
         execute SGn
```

Clause blocks contain clause control instructions which execute the various subgoals, as well as get and put instructions to set up the arguments to each subgoal.

3. Indexing Instructions

Indexing instructions are used to filter the set of clauses that are candidates for unification. By eliminating clauses that cannot possibly unify with the current goal, they improve performance.

Indexing instructions are usually the first instructions in procedures with more than one clause. They examine the first argument of the current goal (which has been placed in register A1 by previously executed code) and branch to an appropriate label, depending on the type of the value in A1.

3.1. switch_on_term

usage: switch_on_term Lc, Ll, Ls

The L's are labels of the blocks of code to execute if the first argument of the current goal is a constant, a list, and a structure respectively. (If the first argument is a variable, all clauses must be attempted, so execution simply falls through to the next statement).

As an example of where this instruction might be used, consider the following Prolog procedure:

```
func(c,X) :- ...
func([a,b], X) :- ...
func(s(1,2), X) :- ...
```

These clauses make up the procedure for "func"; they could be compiled into:

```
func:  switch_on_term        f1,f2,f3
f0:    code to build a choice point to make sure that
         all clauses are tried

f1:    code for first clause
         .
         .
f2:    code for second clause
         .
         .
f3:    code for third clause
         .
         .
```

Let us ignore for the moment the possibility of A1 dereferencing to a variable, and instead suppose the value of A1 is a constant. Then the only clause whose head could possibly unify with it is the first one; hence the switch_on_term instruction will send execution immediately to label f1. Similarly, if A1 refers to a list then only the second clause should be attempted, and if A1 refers to a structure only the third should be tried. Thus the switch_on_term instruction functions like a four-way branch, based on the tag of the dereferenced value of A1.

If A1 is a variable, then all the clauses must be tried; after f1 is attempted, f2 and f3 should be attempted as well. All the switch_on_term instruction has indicated is to fall through to f1. Somehow we need to be able to specify that after f1 is attempted, f2 should be tried, and then f3. This will be taken care of by the "try" and "try-else" instructions, whose functions will be explained shortly.

3.2. switch_on_constant

usage: switch_on_constant N,T

N is the size of an address table located at T. This instruction is used when A1 dereferences to a constant. The value of A1 is hashed into the table T, whose entries point to the appropriate clause to execute. Thus the "switch_on_constant" instruction functions as an N-way branch, based on the constant in A1.

For example, suppose we wished to compile the following Prolog procedure:

```
func(a) :- ...
func(b) :- ...
func(c) :- ...
func(d) :- ...
```

At compile time, the compiler builds the following table at address T:

a	f1
b	f2
c	f3
d	f4

and then generates this block of code:

```
func:   switch_on_constant 4,T

f1:     code for first clause
          .
          .
          .
f2:     code for second clause
          .
          .
          .
f3:     code for third clause
          .
          .
          .
f4:     code for fourth clause
          .
          .
          .
```

In this example the search of the table is assumed to be performed linearly, but of course a hash table could easily be constructed for improved performance. For ease of implementation, the size of the table (N) should be a power of two.

3.3. switch_on_structure

usage: switch_on_structure N,T

This instruction is similar to "switch_on_constant"; the only difference is that it is used when A1 dereferences to a structure. The key used is the principal functor of the structure pointed to by A1. For example, the table and code for the previous example

could have been generated for the procedure:

```
f(a(...)) :- ...
f(b(...)) :- ...
f(c(...)) :- ....
f(d(...)) :- ...
```

with the "switch_on_structure" instruction replacing "switch_on_constant".

Currently, the switch_on_constant and switch_on_strucutre instructions are in a state of flux. Several possibilities exist for the location and format of the hash table, depending on the kind of hardware support provided for hashing. In the present implementation, a simple linear search is performed.

4. Procedure Control Instructions

Procedure control instructions deal with the creation, modification, and deletion of choice points on the control stack. They are the only instructions which do so.

4.1. Try instructions

Try instructions are used when more than one clause is to be attempted. The "try" instruction is used for the first clause to be tried, the "retry" instruction for all clauses in the middle, and the "trust" instruction for the last clause.

4.1.1. try

usage: try L

L is the label of the first clause to try. The try instruction creates a choice point on the stack, saving all the necessary registers and a pointer to the following code, since this is where execution is to resume should the clause that is about to be attempted fail.

4.1.2. retry

usage: retry L

L is the label of the next clause to try. This instruction does much less than "try"; it simply saves a pointer to the following code in the current choice point, overwriting the previous pointer stored there, and continues execution at label L.

4.1.3. trust

usage: trust L

L is the label of the last clause to try. This instruction is so named because it represents the last possible clause in a procedure to attempt; it must be "trusted" to succeed, otherwise the parent goal will fail. Since this instruction is used for the last clause of a set of clauses to be attempted, it will throw away the current choice point.

4.2. Try-else instructions

The "try-else" instructions instructions are simply variants of the "try" instructions. With the "try" instructions, the next clause to execute is specified by the argument L, and the place to return to should that clause fail is the code following the instruction. With the "try-else" instructions, however, the next clause to execute is the

code following the instruction, and the place to return to should that clause fail is given by the argument L.

4.2.1. try_me_else

usage: try_me_else L

L is the label of the clause to attempt should the following code fail. Like "try", "try_me_else" is used for the first clause of a set: it sets up a choice point on the stack.

4.2.2. retry_me_else

usage: retry_me_else L

L is the label of the next clause to attempt should the following code fail. It is used for clauses that are not first or last in a set, just like "retry". It modifies the current choice point by changing its current indicator of where to continue should the current clause fail to L, since L is the label of the next clause to attempt.

4.2.3. trust_me_else

usage: trust_me_else fail

This instruction is used to discard the current choice point on the stack. The argument 'fail' is supplied because if the following clause fails, the parent goal should fail as well. The instruction "trust_me_else fail" discards the current choice point by resetting B and HB, just like the "trust" instruction.

4.2.4. cut

usage: cut

This instruction is used to implement the cut operator (!). In most cases, it simply discards all choice points above the B register value saved in the current environment. However, this may not be satisfactory; if the current procedure has placed a choice point on the stack, then one more choice point must be discarded. This situation is detected through the use of a "cut bit", stored in the most significant byte of the B register value in the current environment. This bit is set when a choice point is placed on the stack, and is cleared by the call, execute, and proceed instructions. Thus the cut instruction can examine the cut bit and discard the correct number of choice points.

4.2.5. cutd

usage: cutd [label]

This instruction throws away all choice points up to and including the one whose L value (see section 2.2.1.2) is equal to the address of [label]. "cutd" it is needed to correctly implement disjunctions.

A disjunction (->) is a common way of implementing if-then statements in Prolog. For example, the Prolog code fragment

(s -> t, u; v)

means that if s succeeds attempt t and u, otherwise attempt v. The compiler generates the correct code for disjunctions by generating a try_me_else instruction to set up a choice point before the "if" part, and then a cutd instruction to make sure the "else"

part is not attempted if the "if" part succeeds. For example, the Prolog clause

```
a(X) :- (s(X) -> t(X), u(X) ; v(X)).
```

is compiled into

```
a:
        allocate
        get_variable  Y1,X1
        try_me_else  _170
        call  s/1,1
        cutd  _170
        put_value  Y1,X1
        call  t/1,1
        put_unsafe_value  Y1,X1
        call  u/1,1
        execute  _171
170:
        trust_me_else  fail
        put_unsafe_value  Y1,X1
        call  v/1,0
_171:
        deallocate
        proceed
```

Note the use of the try_me_else instruction to set up an artificial choice point. Should the call to s succeed, control will proceed to the cutd instruction. This throws away all choice points on the stack, back to and including the artificial choice point. Thus when failure eventually occurs, backtracking will occur to any choice points set up before the clause was entered; the call to v will never be executed. Should the call to s fail, control will resume at label 170: the choice point will be removed, and the goal v will be attempted.

Notice that the semantics of cutd are such that if the "if" part of a disjunction sets up a choice point, it will be discarded. Thus the Prolog program

```
c(a).
c(b).
d(a).
d(b).
e(a).
e(b).

start(X) :- (c(X) -> d(X), e(X) ; fail).
start(X)?
```

will produce as its only solution X = a. This is consistent with the implementation of disjunctions in Cprolog. Fortunately, the most common use of disjunctions is in situations where the condition tested does not set up a choice point; for example, ... X == Y -> dothis ; dothat. So implementing cutd this way isn't a bug; it's a feature.

"cutd" is also needed to correctly implement the cut operator when it occurs *inside* a disjunction. Consider this example:

```
a(b) :- r, (s -> t, !, u; v), w
```

Normally, cut throws away choice points back to the one at the top of the stack before the clause was entered. With this cut, however, we wish to save all choice points present on the stack before the disjunction was entered: in this example the extra choice points would be those associated with r. The compiler handles this using "cutd", compiling the above into:

```
a:
        allocate
        call  b/0,0
        try_me_else  _154
        call  c/0,0
        cutd  _154
        call  d/0,0
        cutd  _154
        call  e/0,0
        execute  _155
_154:
        trust_me_else  fail
        call  f/0,0
_155:
        deallocate
        proceed
```

In essence, "cutd" is a selective cut; it enables the compiler generate code that creates and discards choice points corresponding to known locations in code.

4.2.6. fail

usage: fail

"fail" simply causes goal failure, by invoking the failure subroutine directly. Its operation is described in the chapter on builtin functions.

4.3. Some Examples Using Indexing Instructions

Consider the following Prolog procedure:

```
func(a) :- ...
func(b) :- ...
func(c) :- ...
```

We can compile the control structure of the procedure using indexing instructions as follows:

```
func:
        switch_on_term      fcon,fail,fail
fvar:
        try fa
        retry fb
        trust fc

fcon:
        switch_on_constant N,T

fa:
        code for first clause
        .
        .
        .

fb:
        code for second clause
        .
        .
        .

fc:
        code for last clause
        .
        .
        .
```

assuming that we have built at compile time a table whose entries with keys a, b, and c are fa, fb, and fc, respectively.

Suppose that the A1 dereferences to a variable. The "switch_on_term" instruction will cause execution to fall through to the label "fvar". First, "try fa" will be executed, setting up a choice point and saving a pointer to the following code in that choice point. When the first clause fails, execution will return to the point indicated in the choice point, and the "retry fb" instruction will be executed. It will update the choice point again, and then the code for the second clause will be executed. When this clause fails, control will return to the "trust fc" instruction, which will throw away the choice point the "try fa" instruction had placed on the stack and restore the B and HB registers.

Suppose, however, that the first argument of the current goal dereferences to a constant. Then, the "switch_on_term" instruction would cause execution to proceed to label "fcon". The "switch_on_constant" instruction would be executed, causing control to proceed at whatever label was indicated by the constant table. Since no choice point would be set up, only the appropriate clause would be tried; when it fails the parent goal will fail.

Finally, if A1 dereferences to a list or a structure, the goal fails immediately, as indicated by the last two arguments of the "switch_on_term" instruction.

As another example, consider the heads of clauses for the concatenate procedure (this example is taken from Warren's paper [1]):

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

The control structure for this procedure would be compiled as follows:

```
concat:
        switch_on_term  C1, C2, fail
lv:
        try  C1
        trust C2
C1:
        code for first clause
C2:
        code for second clause
```

Notice that if only one clause is to be tried (in this example, if A1 dereferences to a list or a constant), the labels and instructions have been arranged in such a way that a choice point will never be built. A choice point will only be constructed if A1 dereferences to a variable, in which case we wish to try both clauses.

If A1 is a variable, control will fall through to lv. The first clause will be executed after a choice point has been built and a pointer to the following code has been saved in it. When the first clause fails, execution will resume at the "trust C2" instruction. This instructions will throw away the current choice point, since no clauses remain to be tried. Thus if the second clause fails, backtracking will occur to the most recent active choice point.

If instead of a variable A1 is a constant, then control will branch to the first clause, without setting up a choice point. (We wish to check the first clause if A1 dereferences to a constant because the constant could be the special constant NIL, indicating a nill list). Similarly, if A1 dereferences to a list, then only the second clause will be attempted.

5. Clause Control Instructions

Control instructions are responsible, as Warren says, for the control transfer and environment allocation associated with procedure calling [1]. There are five clause control instructions: three deal with control transfer ("proceed", "execute", and "call"), and two deal with environments (the "allocate" and "deallocate" instructions),. The first three instructions correspond to the three possible control transfers that may occur when a goal succeeds. After a goal succeeds, we will have one of the following cases: 1) the clause was a unit clause, with no body, 2) the clause has one goal that remains to be executed, or 3) the clause has more than one goal that remains to be executed. The appropriate actions to take in each case are implemented by the "proceed", "execute", and "call" instructions, respectively.

5.1. proceed

usage: proceed

The "proceed" instruction is used at the end of a unit clause. After unification with a unit clause, the parent goal has succeeded, and we wish to continue at the point indicated by the CP register. Thus all the "proceed" instruction does is to load P with the contents of the CP register.

For example, consider the compilation of the following procedure:

```
lt3(0).
lt3(1).
lt3(2).
```

The resulting code would look like:

```
lt3
  switch_on_term lt3con, fail, fail

lt3var:
      try lt3a
      retry lt3b
      trust lt3c

ltcon:
      switch_on_constant 4, Table
lt3a:
      code for first unit clause
      .
      .
      proceed

lt3b:
      code for second unit clause
      .
      .
      proceed

lt3c:
      code for third unit clause
      .
      .
      proceed
```

We assume that the table stored at Table has been appropriately constructed to indicate the appropriate branch addresses.

All the clauses end with "proceed", because all the clauses are unit clauses. If all the associated unifications are successful, then the parent goal has succeeded, and we want to "proceed" on to the next goal (whose location is indicated by the CP register).

5.2. execute

usage: execute Proc

The "execute" instruction appears as the last instruction in the final goal in the body of a clause. It is the equivalent of a goto statement, whose label is given by "Proc".

For example, the Prolog clause

```
back(X,Y) :- front(Y,X).
```

would be compiled as

```
back:  code for matching A registers
       .
       .
       code for preparing A registers for procedure "front"
       .
       .
       execute front
```

5.3. call

usage: call Proc,n

The call instruction is used to invoke a goal when more than one goal remains in the body of the clause being executed. Proc is the name of the procedure to call, and n is the number of variables in the current environment. It sets CP to point to the following code, sets N equal to n and sets P equal to the address indicated by Proc.

For example, the Prolog clause

```
a(X,Y,2) :- b(Y,X), c(Y), d(Y).
```

would be compiled as follows:

```
a:     allocate
       code for matching the A registers
       .
       code for preparing the A registers for procedure "b"
       .
       call b,1
       .
       code for preparing the A registers for procedure "c"
       .
       call c,1
       .
       code for preparing the A registers for procedure "d"
       .
       execute d
```

(The "allocate" instruction in this code will be explained in the next section).

5.4. allocate

usage: allocate

The allocate instruction is used at the beginning of a clause with more than one goal in the body. It saves the E, CP, N, and B register values in the current environment on the stack, and allocates space for the new environment of the current clause.

For example, the Prolog clause

```
a(X,Y,Z) :- b(X), c(X,Y), d(Z).
```

would have "allocate" as its first instruction:

```
a:    allocate
      code for the clause
      .
      .
      .
```

whereas the clause

```
a(X,Y) :- b(Y,X).
```

would not have an allocate instruction. Since this clause has only one goal in its body, an environment is not needed.

5.5. deallocate

The deallocate instruction is used to remove the environment from the stack when it is no longer needed. Warren's PLM employs last call optimization, which means that environments are discarded not when the last goal in the body of a clause has finished executing, but when the *next to last* goal in the body of a clause has finished executing. This can be done because when only the last goal remains to be executed, the saved state information stored in the environment is no longer needed, and can be discarded. This saves space on the stack. One problem with last call optimization which will be examined in more detail later is the problem of dangling references: by throwing away the environment before the last clause has executed, you may be discarding a variable that the last clause will access; that clause will then have a reference to storage that has been deallocated. Such variables are referred to by Warren as "unsafe" variables. Warren remedies this problem with the "put_unsafe_value" instruction, a member of the class of put instructions which will be discussed shortly. For now, it suffices to remember that environments can be discarded as soon as the next to last goal in the body of a clause is executed.

For example, the Prolog clause

```
a(X,Y,Z) :- b(X), c(X,Y), d(Z).
```

would have a "deallocate" instruction right before the call to for "d", thus:

```
a:    allocate
      code for matching A registers
      .
      code for preparing for call to procedure b
      .
      call b,3
      code for preparing for call to procedure c
      .
      call c,2
      code for preparing for call to procedure d
      .
      deallocate
      execute d
```

5.6. Environment Trimming

One of the optimizations of Warren's machine machine is called environment trimming. This refers to the assignment of permanent variables to locations in their environment by the compiler. The compiler attempts to allocate permanent variables in

such a way that the ones near the top of the stack can be discarded as soon as possible. This saves stack space.

Environment trimming is implemented on the PLM using the N register and the call instruction. The N register holds the number of permanent variables in the current environment. It is kept up to date by the call instruction, which loads the N register from its second argument. The contents of the N register are used whenever an environment is on top of the stack, and a new environment or choice point is to be built. The starting location of the new environment or choice point is calculated as the contents of the E register plus 4 (for the CP,B,H, and E values in an environment), plus the contents of the N register.

For example, the Prolog clause

```
a(X,Y,Z,W) :- b(W), c(Z), d(Y), e(X).
```

would be compiled into

```
a:
        allocate
        get_variable  Y1,X1
        get_variable  Y2,X2
        get_variable  Y3,X3
        put_value  X4,X1
        call  b,3
        put_unsafe_value  Y3,X1
        call  c,2
        put_unsafe_value  Y2,X1
        call  d,1
        put_unsafe_value  Y1,X1
        deallocate
        execute  e
```

Here, the compiler has assigned X to Y1, Y to Y2, and Z to Y3. When 'b' is called, the environment of this clause has three permanent variables. Once b finishes, and the registers are prepared for the call to can be reclaimed. Thus when 'c' is called, the second argument of the call instruction is '2', since now the environment has only two permanent variables.

Of course, most goals in a clause share variables, so this example is not typical. Nonetheless, the compiler will always allocate variables in such a way as to allow the maximum amount of environment trimming inherent in a clause.

6. Get instructions

Get instructions are used to unify the contents of the A registers against whatever arguments are appropriate for the head of the current clause being executed. All get instructions take the form "get_.... [dest,] source". The source is always an argument register, whose dereferenced value is to be either unified with "dest" or simply stored there, depending on the instruction. The "dest" argument may be implicit or unnecessary; this too depends on the instruction.

The get instructions are:

get_constant
get_nil
get_value
get_variable
get_list
get_structure

6.1. get_constant

usage: get_constant c,Ai

This instruction dereferences Ai, and tries to unify the result with the constant "c". If the unification succeeds, control proceeds on to the next instruction. Otherwise, the goal fails and backtracking occurs.

This instruction is used whenever an argument in the head of a clause is a constant. For example, the following Prolog clause

```
a(red,white,blue) :- b(...).
```

would be compiled into:

```
a:    get_constant red, A1
      get_constant white, A2
      get_constant blue, A3
      code to prepare the A register for procedure "b"
      .
      execute b
```

First, the A1 register would be dereferenced and its contents unified with the constant "red". If this unification was successful, then the remaining two arguments will be checked for unification with "white" and "blue". Only if these unifications are successful will procedure "b" be called; otherwise the goal will fail. Thus the body of this clause will only be executed if the first three arguments of the current goal can unify with "red", "white", and "blue", which is just what we desire.

6.2. get_nil

usage: get_nil Ai

The "get_nil" instruction is really just a special case of the "get_constant" instruction; it is semantically identical to "get_constant [], Ai". Since the special constant NIL, signifying a null list, is used so often, a separate instruction is used for dereferencing an argument register and unifying it with NIL. For example, if the previous example were modified slightly,

a(red,[],blue) :- b(...).

we would obtain the following compiled version:

```
a:    get_constant red, A1
      get_nil A2
      get_constant blue, A3
      code for preparing the A registers for procedure "b"
      .
      execute b
```

6.3. get_variable

usage: get_variable [A | X | Y]n, Ai

This instruction simply transfers the contents of Ai to An, Xn, or Yn. If the first argument is An or Xn, the contents of Ai are transferred to argument register n. If, however, the first argument is Yn, then the contents of Ai are transferred to a location at offset n in the current environment on the stack. This is done to initialize permanent variables, mentioned briefly when first discussing environments. Each permanent variable is given a unique offset n in the environment of the current clause, and is referred to by PLM instructions as Yn.

As one example of "get_variable" , consider the following simple clause:

a(2,X) :- b(X).

This could be compiled into:

```
a:    get_constant 2
      get_variable A1, A2
      execute b
```

Notice that no unification was performed against the contents of A2, because anything can unify with a variable.

As a more complicated example, illustrating permanent variables, consider the following:

a(2,X,Y) :- b(X), c(Y).

Before we call b, we must save the contents of A3 (the variable Y) in a permanent variable, since it will be needed to set up the call to procedure "c". Thus we wish to produce the following code:

```
a:    allocate
      get_constant 2, A1
      get_variable A1, A2
      get_variable Y1, A3
      call b,1
      .
      load A1 with contents of Y1
      .
      deallocate
      execute c
```

6.4. get_value

usage: get_value [A | X | Y]n, Ai

The "get_value" instruction dereferences Ai, and then attempts to unify the result with the contents of An, Xn, or Yn, where An, Xn and Yn all have the same meanings as before. It is used for arguments in the head of a clause that represent variables that have already occurred at least once in the head of the clause. For example, the unit clause

```
a(X,X).
```

would be complied into

```
a:    get_value A2, A1
      proceed
```

Here we dereference A1 and see if the resulting value can unify with the contents of A2. Only if this unification is successful will the clause succeed. Notice that

```
a:    get_value A1, A2
      proceed
```

would do the same thing; either compilation is correct.

Here is a more complicated example:

```
a(X,Y,X,Y) :- b(X,Y), c(X).
```

would be compiled into

```
a:    get_variable Y1, A1
      get_value Y1, A3
      get_value A2, A4
      code for body of clause
      .
```

6.5. The 'read' and 'write' Unification Modes of the PLM

Analysis and explanation of the remaining get instructions will be made easier by first examining an important feature of the PLM: unification modes. This concept was first used by Warren to implement structure copying, and is employed in the PLM for similar reasons.

Consider a program containing the unit clause "t([1,2,3]).", and the query "t(X)?". When the unbound variable X is unified with the list [1,2,3], we wish the unification to succeed, with X being bound to a new copy of the list on the heap. Consider what happens, however, if instead of an unbound variable in A1 we have a list, as in the query "t([1,2,4])?". In this case, we wish to traverse each element of the list, checking that each element of the list in the query matches each element of the list in the clause in the program. So if unification is being performed against existing structured data as opposed to an unbound variable, the actions to be taken during unification are very different. In the case of an unbound variable, we wish to build a new list or structure on the heap and change the argument register to point to it. In the case of compound data, we wish instead to check the corresponding elements of the list or structure, and attempt to unify them with each other.

These two different courses of action correspond to the two unification modes of the PLM. Since we cannot know at compile time what the data types in the argument

registers will be when a procedure is called, the compiler cannot generate different code for the two cases. The type of unification to be done must be detected at runtime. In particular, it is done by certain instructions of the PLM. When any of these instructions are executed, the appropriate argument register is dereferenced, and its tag is examined. If this tag is a variable then unification proceeds in "write" mode. This mode is called "write" mode because a new list or structure is written onto the heap and the previously unbound variable in the argument register is bound to it. If, however, this tag is a list(structure), then unification proceeds in "read" mode. This mode is called "read" mode because unification is performed on each element of the compound data separately; the two data items are "read" and compared against one another. In the PLM, the individual elements of the structure being unified against are addressed via the S register.

Once the mode is set by an instruction it remains set until changed by another instruction. Instructions that perform different actions depending on the unification mode check the current mode under which unification is proceeding and perform the appropriate action. This will be examined in more detail when the "unify" family of instructions is discussed.

6.6. get_list

usage: get_list Ai

"get_list" dereferences Ai, and examines the tag of the result. If Ai dereferences to a list, then the S register is set to point to the first argument of the list, and unification proceeds in "read" mode. If, however, the result is a variable, then that variable is bound to a new list pointer on top of the heap, and execution proceeds in "write" mode.

"get_list" is used whenever an argument in the head of a clause is a list. For example, the unit clause

```
a([b,c,d]).
```

would be compiled into

```
a:    get_list A1
      instructions to perform unification with the elements
        of the list '[b,c,d]'
      .
      .
      proceed
```

6.7. get_structure

usage: get_structure F, Ai

"get_structure" functions in the same manner as "get_list"; the only difference is that unification will proceed in "read" mode if Ai dereferences to a structure with principal functor F. For example, the unit clause

```
a(2, q(x,y,z)).
```

would be compiled into

```
a:    get_constant 2, A1
      get_structure q, A2
      instructions to perform unification with the elements
            of the structure 'q(x,y,z)' (to be explained shortly)
            .
      proceed
```

7. Put instructions

"put" instructions are used to move data into the A registers. These instructions are used only for data movement; no unification is ever performed. Put instructions take the form "put_... [source,] dest", where "dest" is an argument register and "source" is either Xi, Yi, or a constant. The word "source" should be interpreted loosely, however; some put instructions initialize the "source" operand from another implicit location before transferring the contents of "source" to "dest".

The "put" instructions are:

put_constant
put_nil
put_value
put_variable
put_list
put_structure
put_unsafe_value

7.1. put_constant

usage: put_constant c, Ai

"put_constant" simply puts the constant c into argument register Ai. It is used whenever an argument of a goal in the body of a clause is a constant. For example, the clause

```
a(c) :- f(a), g(b,d).
```

would be compiled into:

```
a:    allocate
      get_constant c, A2
      put_constant a, A1
      call f,0
      put_constant b, A1
      put_constant d, A2
      deallocate
      execute g
```

7.2. put_nil

usage: put_nil Ai

"put_nil" is related to "put_constant" in the same way that "get_nil" is related to "get_constant": "put_nil" has an implicit constant argument of '[]'. It is used whenever an argument of a goal in the body of a clause is the empty list. For example:

```
a(X, Y) :- f([],a,b).
```

would be compiled into:

```
a:    put_nil A1
      put_constant a, A2
      put_constant b, A3
      execute f
```

(Notice that in this example no get instructions are needed to match against X and Y, since any two arguments will match them and they are not used in the body of the clause).

7.3. put_value

usage: put_value [A | X | Y]n, Ai

"put_value" takes the value of An, Xn, or Yn and stores it in Ai. It is used for arguments of goals in the body of a clause that are bound variables. For example, the clause

```
a(X,c) :- b(d,X), c(X).
```

could be compiled into

```
a:    allocate
      get_variable Y1, A1
      get_constant c, A2
      put_constant d, A1
      put_value Y1, A2
      call b,1
      put_value Y1, A1
      deallocate
      execute c
```

First, the permanent variable X is saved at offset 1 in the environment of the clause. Next, A2 is unified with the constant c, and A1 is loaded with the constant d. We then wish the value of X (since X may be bound) to be loaded in A2, so the "put_value" instruction is used to transfer the value of the permanent variable X back into A2. After calling b, we use the instruction again to load A1 before we deallocate and execute the code for "c".

7.4. put_variable

usage: put_variable [A | X | Y]n, Ai

"put_variable" is used for arguments of goals in the body of the clause that are unbound variables. If the "source" argument is Yn (a permanent variable), then Yn is initialized to an unbound variable, and An is set to point to Yn. If the "source" argument is Xn or An, then the instruction creates an unbound variable on the heap and puts a reference to it in registers Xn and Ai.

For example, the Prolog clause:

```
f(c,X) :- g(d,X,Y).
```

would be compiled into:

```
f:
      get_constant c, A1
      put_constant d, A1
      put_variable Y1, A3
      execute g
```

Since the variable Y is unbound and permanent, we use the "put_variable" instruction to save it in the current environment and to load the appropriate argument

register with a reference to it.

7.5. put_list

usage: put_list Ai

"put_list" is used for arguments of goals in the body of a clause that are lists. It places a list pointer that points to the top of the heap and sets the unification mode to "write". "put_list" and "put_structure" are the only put instructions that set the unification mode.

For example, the clause

```
f(X) :- a(Y, Y, [1,2,3]).
```

would be compiled into:

```
f:    put_variable X4, A1
      put_value X4, A2
      put_list A3
      "unify" instructions to create [1,2,3],
            to be explained shortly
      .
      execute a
```

The clause

```
f(X) :- a([1,2,3]), b([a,b,c]).
```

would be compiled into:

```
f:    allocate
      put_list A1
      "unify" instructions to create '1,2,3'
            (to be explained shortly)
      .
      call a,0
      put_list A1
      "unify" instructions to create '[a,b,c]'
            (to be explained shortly)
      .
      deallocate
      execute b
```

7.6. put_structure

usage: put_structure F, Ai

"put_structure" is related to "put_list" in the same way that "get_structure" is related to "get_list". It is used for arguments to goals in the body of a clause that are structures with principal functor F. "put_structure" pushes the functor F onto the heap and places a structure pointer to it in register Ai. Execution then proceeds in "write" mode.

For example, the clause

```
a(X) :- b(q(g,h,r(i,5))), c(X).
```

would be compiled into:

```
a:    allocate
      get_variable Y1, A1
      put_structure q, A1
      "unify" instructions for building 'q(g,h,r(i,5)))'
        (to be explained shortly)
      .
      call b,1
      put_value Y1, A1
      deallocate
      execute c
```

7.7. put_unsafe_value

usage: put_unsafe_value Yn, Ai

Recall that the PLM employs a generalized form of tail recursion optimization by deallocating the environment of the current clause before the last goal in the body of the clause is executed. If, however, at the time of deallocation variables in the last goal reference the current environment, then these variables will be dangling references. Thus, when setting up registers for the last goal in a clause, we have potential dangling references: if a variable was instantiated in the body of the clause (that is, it was created with a 'put_variable' instruction), then it may be unsafe. We must check to see if it dereferences into the current environment, and if so we must 'globalize' the variable by moving it from the stack to the heap.

This is just what the 'put_unsafe_value' instruction does. This instruction dereferences its first argument Yn. If the result is a variable bound to a location outside the current environment then the variable is safe after all: the instruction acts just like "put_value". If, however, the variable dereferences to the current environment, then a new variable is created on the top of the heap, and Ai is set to point to it.

Consider the following clause:

```
a(X) :- b(X,Y),c(Y).
```

Here, the variable Y is potentially unsafe. Thus, before the call to c, A1 is loaded with Y with a put_unsafe_value instruction, as part of the following compilation:

```
a:
      allocate
   put_variable Y1, A2
   call b,1
   put_unsafe_value Y1, A1
   deallocate
   execute c
```

8. Unify instructions

The last family of instructions to consider are the "unify" instructions. These instructions are the only ones that make use of the "read" and "write" unification modes. They are used for unifying against the elements of existing structures, and for creating new structures. The "unify" instructions are:

unify_constant
unify_value
unify_variable
unify_void
unify_cdr
unify_nil

When working with a list, the first four instructions work with the "car", while the last two work with the "cdr".

9. unify_constant

usage: unify_constant c

These instructions are used for elements of lists or structures that are constants. If unification is proceeding in "read" mode, then the term pointed to by the S register is dereferenced, and the result is unified against the constant c. If unification is proceeding in "write" mode, then the constant c is pushed onto the heap. In either case, the S register is set to point to the next item in the structure.

9.1. unify_value

usage: unify_value [A | X | Y]i

"unify_value" is used for elements of structures that are bound variables. If "read" mode is being used, the contents of the word pointed at by the S register are unified with the dereferenced value of Xi, Yi, or Ai. In "write" mode, the dereferenced value of the specified register is pushed onto the heap. If the argument was not Yi, the dereferenced result is left in the register specified.

For example, the Prolog clause

a(Q) :- b([1,Q,2])

would be compiled into

```
a:    get_variable X2, A1
      put_list A1
      unify_constant 1
      unify_value X2
      unify_constant 2
      unify nil
      execute b
```

Note the use of the register X2 as a temporary storage location for saving the variable Q; without it Q would be destroyed as we built the list in preparation for the call to b. Since the clause had only one goal in the body, an environment and a permanent variable location for Q was not needed.

(The unify_nil instruction will be explained shortly).

9.2. unify_variable

usage: unify_variable [A | X | Y]i

"unify_variable" is used for elements of structures that are unbound variables, *and that occur more than once in the clause.* (Variables that violate this second restriction are represented by the "unify_void" instruction, to be considered next). If unification is proceeding in "read" mode, then the contents of the data word pointed at by the S register are simply transferred to Xi or Yi or Ai. If "write" mode is being used, then a new unbound variable is pushed onto the heap, and a reference to it is stored in Xn, An, or Yn.

For example, the clause

```
a(Y) :- b([X,Y,X]).
```

would be compiled into

```
a:    get_variable X2, A1
      put_list A1
            unify_variable X3
            unify_value X2
            unify_value X3
            unify_nil
      execute b
```

9.3. unify_void

usage: unify_void n

"unify_void" represents n consecutive elements of a structure or list that are single occurrence variables. If "read" mode is being used, then the next n elements of the structure being addressed via the S register are simply skipped. If "write" mode is being used, then n new unbound variables are pushed onto the heap.

This instruction is not strictly essential to the completeness and correctness of the PLM instruction set. The main difference between "unify_void" and "unify_variable" is that "unify_variable" requires an argument in the form of a register name in which to place a result. The following two sequences are equivalent:

```
unify_void n

unify_variable Ai
unify_variable Ai
(n times)
```

provided that none of the A registers used by the "unify_variable" instructions are ever used in the rest of the clause. Using "unify_void" is, however, simpler and faster. For example, the clause

```
a(c,X) :- b([Q,R,c,S,c,T]).
```

would be compiled into

```
a:    get_constant c, A1
      put_list A1
            unify_void 2
            unify_constant c
            unify_void 1
            unify_constant c
            unify_void 1
            unify_nil
```

Notice that had any of the variables in the list been needed by other goals in the body, then "unify_variable" would have had to have been used; "unify_void" does not save pointers to any of the unbound variables it creates. For example, consider

```
a(c,X) :- b([Q,R,c,S,c,T]), c(T), d(X).
```

We would have to compile this into:

```
a:    allocate
      get_constant c, A1
      get_variable Y1, A2
      put_list A1
            unify_void 2
            unify_constant c
            unify_void 1
            unify_constant c
            unify_variable Y2
            unify_nil
      call b,2
      put_unsafe_value Y2, A1
      call c,1
      put_value Y1, A1
      deallocate
      execute d
```

Note the use of "unify_variable" instead of "unify_void" for the variale T.

9.4. unify_cdr

usage: unify_cdr [A | X | Y]i

This instruction is not mentioned in Warren's original paper. It was included in the PLM instruction set to handle the unification and binding of the tail of a list. Prolog allows the tail of a list to be a variable, as for example the variable Y in "a([X | Y])". Should a be called with the argument [1,2,3,4], we would like X to be bound to '1' and Y to be bound to [2,3,4]. Traversing the list and using a "unify_variable" instruction for Y will not work here, because we wish Y to be unified with the entire remainder of the list, (the cdr), and not simply the next entry '2'.

"unify_cdr" handles this situation. In "write" mode it creates an unbound variable in the cdr field of the list being built and stores a pointer to it in Xi, Yi, or Ai. In "read" mode, if the cdr of the list being traversed is an unbound variable, then the register specified by the argument field is set to point to it. Otherwise the register specified is set to a list pointer that points to the location addressed by the S register.

For example, the clause

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

would be compiled into

```
concat: get_list A1
            unify_variable X4
            unify_cdr A1
        get_list A3
            unify_value X4
            unify_cdr A3
        execute concat
```

10. unify_nil

usage: unify_nil

The "unify_nil" instruction is used to unify data elements with the special constant NIL. It is similar to the get_nil and put_nil instructions. In read mode, the location addressed by the S register is unified against the constant NIL. In write mode, the list being built is "closed off" by writing the constant NIL into the last location on the heap.

To see how this instruction is used, consider the following example:

```
a([c,d,e]) :- b([f,g,h]).
```

This clause would be compiled into

```
a:    get_list A1
            unify_constant c
            unify_constant d
            unify_constant e
            unify_nil
      put_list A1
            unify_constant f
            unify_constant g
            unify_constant h
            unify_nil
      execute b
```

11. Dynamic switching from read to write mode

Under certain conditions, the unification instructions that work with the 'car' can change the unification mode from 'read' to 'write'. This can happen if a list built at runtime with an unbound cdr is unified against a compiled list. For example, consider the unit clause a([1,2,3,4]). It would be compiled into:

```
a:
  get_list A1
       unify_constant 1
       unify_constant 2
       unify_constant 3
       uinfy_constant 4
       unify_nil
```

Suppose we asked the query a([1,2 | X])? The first instruction, get_list, would set the unification mode to read, since the tag of the first argument is a list. The next two unify_constant instructions would succeed. However, in order for the code to work correctly X must be unified to the remainder of the list, [3,4]. The next unify_constant instruction detects this situation by switching the unification mode to 'write' if its argument is an unbound cdr. Thus "unify_constant 3" would change the unification mode, as well as create a new list with first element 3, bound to X.

All the unify instructions that work with the car check for arguments that are unbound cdrs, and change the unification mode from read to write if this is the case. These are the only instructions that do so. The kind of situation outlined above is the only one in which the unification mode is changed "dynamically", by unify instructions. In all other cases, the unification mode is set by the get or put instructions.

CHAPTER 3

Other Fundamental PLM Operations

1. Other Fundamental Operations of the PLM

Several of the PLM instructions perform one or more of a set of basic operations not visible at the ISA level. These operations are:

dereference
decdr
unify
bind
trail
fail

1.1. Dereference

This is just the usual dereferencing operation. It takes one argument; if the argument is a variable, it returns the data item at the end of its pointer chain. (Dereferencing an argument that is not a variable returns the argument). All unification is performed on dereferenced arguments.

It should be noted that while it is possible for the PLM to produceereference chains of length greater than one, in practice virtually all dereferencing involves one level of indirection. This is because of the way variables are bound, as first suggested by Warren [1]: when two variables are bound to one another, the newer is always bound to the older, always binding the fully dereferenced value.

1.2. Decdr

Decdring is a fundamental operation implemented to support cdr-coding. It is used to fetch the next element from a list. If the cell to be decdr'd has its cdr bit set and is of type 'list', then the S register is set to point to the cell addressed by the current cell, this value is returned, and the S register is incremented. Otherwise, if the cdr bit is set and the cell is a constant, the special constant "NIL" is returned, and the S register is incremented. If neither case applies, then the value to be decdr'd is returned unchanged.

1.3. Unify

The unification routine is called by the get and unify instructions. It takes two arguments and attempts to unify them, returning either success or failure, according to the following algorithm:

```
unify(arg1, arg2)

IF types of arguments different
  return(FAILURE)
ELSE IF either argument is a variable
  {
  IF both arguments are variables
       bind newer to older
  ELSE
       bind variable argument to other argument
  return(SUCCESS)
  }
ELSE IF both arguments are constants
  {
  IF both arguments are equal
       return(SUCCESS)
  ELSE
       return(FAILURE)
  }
ELSE IF both arguments are lists OR both arguments are structures
  {
  FOR (each element in list/structure)
       IF (not(unify(el't arg1, el't arg2))
       return(FAILURE)
  return(SUCCESS)
  }
```

Unification failure will always cause goal failure; see below. The unification routine may also call the bind routine.

1.4. Bind

The bind routine takes two arguments, an address and a data value, and stores the data value at the given address. The bind routine calls the trail routine, to see if the binding should be trailed.

1.5. Trail

The trail routine examines the address passed to it from "bind". If it is a stack location and it is behind the current value in the B register, then it pushes the address onto the Trail and increments the TR register. Likewise, if the address is in the heap and it is behind the current value in the HB register, it will also be pushed onto the Trail and the TR register will be incremented. If neither case holds, then the binding is not trailed.

1.6. Goal Failure

If unification fails, the fail routine is called. Goal failure restores the state of the machine at a previous point in the computation. First, all locations on the Trail, from the value of TR saved in the current choice point to the current value of TR are reset to unbound variables. Next, the argument registers and registers E,CP,H,P, and N are all reloaded from the current choice point. The PDL is reset, and control proceeds to the next instruction indicated by the P register.

2. References

1. D.H.D. Warren, *An Abstract Prolog Instruction Set,* SRI International, Menlo Park,CA (1983). Technical Report

2. Tep Dobry, A. M. Despain, and Yale Patt, "Design Decisions Influencing the Microarchitecture For A Prolog Machine," *MICRO 17 Proceedings*, (Oct, 1984).

3. Tep Dobry, Alvin Despain, and Yale Patt, "Performance Studies of a Prolog Machine Architecture," *Proceedings of the 12th Int. Symp. on Computer Arch.*, pp. 180-190 IEEE Computer Society Press, (1985).

4. Tep Dobry, *PLM Simulator Reference Manual,* Computer Science Division, University of California, Berkeley, CA (July, 1984). Technical Note

5. Evan Tick and David Warren, *Towards a Pipelined Prolog Processor,* SRI International, Menlo Park, CA (Aug, 1983). Technical Report

6. Peter Van Roy, *A Prolog Compiler for the PLM,* University of California, Berkeley, CA (Aug, 1984). Master's Report

APPENDIX A

The Set and Access Builtin Predicates

1. Set and Access

Currently, the Prolog functions "assert" and "retract" are not fully implemented. Instead, simpler functions have been designed which permit the global assertion of unit clauses. This mechanism is implemented through the use of a special area called the global heap, which up to now we have not discussed.

The global heap is addressed by a special register called H2; it it similar in function to the H register on the regular heap. The global heap occupies 64 locations in memory, the first sixteen of which may be used as global variables. To set one of them, the internal builtin "set" is provided. "set" takes two arguments: an A register and a global variable number. It unifies the designated global variable with the contents of the argument register, performing the unification on the global heap. To access the new global structure that results, the internal builtin "access" is provided. "access" also takes two arguments: an A register and a global variable number. It attempts to unify the dereferenced value of the indicated global variable with the contents of the specified A register.

For example, the Prolog clause

```
main :- assert(cost(20)), cost(X).
```

would be compiled into:

```
main:
        get_constant 20,X1
        get_constant X2,0
        escape  set
        put_variable  X1,X1
        execute cost

cost:
        get_constant X2,0
        escape access
        proceed
```