1. Introduction.

Meta-programming is a problem solving technique in which a problem is expressed partly at an **object level** (which domain is the intended interpretation of the actual problem), partly at a **meta-level** (which domain ranges over the syntactical expressions of the object domain). Therefore the problem at hands can be conveniently solved, at least in part, by mean of syntactical manipulations at the meta-level on object-level entities [Cerv91, Ster, AiLe]. Most programs only have an object level since they can directly express a problem and its solution. Meta-programs add a limited introspection capability [MaNa]. A broad definition of a meta-program is:

A **meta-program** is a program that treats other programs as data.

According to this definition compilers, interpreters, debuggers and even editors are all meta-programs even if they do not cooperate directly in solving the object problem.

The origin of meta-programming is found in mathematical logic; in particular in incompleteness theorems of Kurt Gödel (the interested reader can refer to [Klee, Ende]).

Sixty years later, meta-programming is a well established technique and it is currently used in several application areas. Due perhaps to its close correspondence with logic, this technique has proven to be useful in theorem proving. One can write a theory simulator as a meta-program and represent axioms and inference rules at the object level [Weyh, AiSC, Cerv91].

Meta-programming is widely used in Artificial Intelligence, in particular in the subfield of Knowledge Based Systems construction.

> "... the integration of knowledge and metaknowledge seems to be a powerful abstraction mechanism that allows to cope with a variety of representation problems, thus providing an alternative to the development of special purpose languages." [AiLe]

Expert knowledge can be conveniently expressed at the object level while the meta-level takes care of performing deductions and supports the typical functionalities of an expert system shell, like explanations, hypothetical reasoning, non-monotonous reasoning, etc. Moreover, meta-programming can be a useful way of structuring the overall knowledge of a system into separated knowledge bases, each ranging over a logically distinct domain, and of managing their mutual interactions. For examples and a further understanding of the role of meta-programming in Knowledge Bases Systems, refer to [AiLe, Ster, StSh, StBe, Bowe].

Meta-programming has also proved useful in implementation of Integrated Programming Environments, especially for non-conventional programming languages, such as Prolog [MaRo, Ross89, Ross90a]. Meta-programming techniques have also been used to enhance the operational semantics and the expressiveness of programming languages like Prolog [GaLa, AiCS, BoWe, Ross89, Ross90a, Ross90b].

Because of the semantic equivalence of all general purpose programming languages, a well-known result of the computability theory, any such language has meta-programming capabilities. However, since a meta-program is supposed to treat other programs as data, only those programming languages that represent programs and data in a homogeneous manner can be used as **meta-**

programming languages. Functional and logic languages have this characteristic: a LISP program is an S-expression and a Prolog program is a term.

Prolog has been used for a long time as a meta-programming language [StSh, Ster, StBe]. Actually, a full meta-interpreter for pure Prolog is just three lines long:

```
solve(true).
solve(A&B) :- solve(A), solve(B).
solve(A) :- clause(A,B), solve(B).
```

Nevertheless, the use of Prolog as a meta-programming language for non trivial programs reveals flaws and limitation in the effective meta-level facilities offered by that language [Lloy88, HiLl88]. Therefore, in the last ten years, a number of extensions or completely new definitions of Prolog have been proposed in order to add sound meta-programming capabilities.

The problem of meta-programming in the context of logic programming was systematically faced for the first time by Bowen and Kowalski in [BoKo] where many important theoretical guidelines were established. Only three years later, the first **logic meta-programming language**, called MetaProlog, was presented to the scientific community [Bowe, BoWe]. Perhaps because MetaProlog had even more flaws than Prolog itself [HiLl88], the research continued and other languages have been proposed [CoLa, CoDL, Cost, HiLl91].

Unfortunately, even though most of these proposals contain interesting and very promising ideas, none provide all of the following capabilities:

- being able to treat any syntactic entity of the language, from the symbols up to the programs at the meta-level;
- allowing an efficient implementation, in both time and space;

- possessing a sound and complete logical semantics for the whole language, and in particular for the meta-level features;
- offering the users simple, powerful and easy-to-use meta-programming facilities.

Therefore, in 1991, at the University of Udine, Italy, an attempt was made to define a logic programming language that includes all the meta-programming capabilities above. The language is called ’Log (read *quote log*) [Cerv91, CeRos91a, CeRo91b, CeRo92, Ross89, Ross90a].

’Log is a pure logic programming language augmented with a mechanism for accessing, decomposing and manipulating the syntactical objects of the language itself. It has been designed to allow the manipulation of the full range of its own syntactical structure, from single characters to complete programs. Moreover the apparent simplicity of the language has played an important role in proving its semantic properties, particularly its soundness and completeness properties.

What was lacking so far to ’Log was a full-range implementation, giving in that way the language a concrete dimension. Actually two attempts have been made. The first one is described in [Ferr] and it illustrates a Modula 2 interpreter for the topmost representation layer of ’Log (programs only were dealt with at the meta-level). The second is a raw prototype written in Prolog by G.F. Rossi in his free time. It has been described to who writes by means of private communications [Ross91] and seems to cover most of ’Log’s aspects. The next step was to write a complete ’Log environment where ’Log programs could have been developed. The heart of the system should have been an executer for the complete language, an implementation of what has been described in [Cerv91]. This developing environment and in particular the implementation of the executer are the subject of this work.

At present, there are two well-established techniques for implementing logic programming languages like Prolog: interpretation and compilation. Early Prolog implementations employed interpreters [KlSz]. In 1983, David Warren proposed a scheme for compiling Prolog into an assembly-like language and then directly executing the resulting instructions [Warr]. This run-time architecture is known as the *Warren Abstract Machine* (**WAM**). Most commercial Prolog systems include are WAM-based. Prolog machines have been implemented both at the microcode and at the hardware level and all of them rely on the WAM design. The boost in efficiency justifies the complex design and implementation of the compiled approach.

Although many Prolog implementations are based on the WAM, it is still poorly understood, mainly because of its intrinsic complexity and to the lack of a clear description of its philosophy and instruction set. Things have somewhat changed last year thanks to Hassan Aït-Kaci after the publication of [AïtK91a, AïtK91b]. The WAM is still the subject of a lot of research: [Bach, BBCT, Buet, Carl, DeMC, Kurs, LiOK].

In this thesis, a WAM-based architercture and programming environment for ’Log will be described. The efficiency of the resulting system is comparable to that of a commercial Prolog compiler, when non meta-level features are used. No measurements concerning the meta-level aspects have been attempted.

This thesis is structured in the following way. Chapter 2 describes the Warren Abstract Machine. Chapter 3 presents ’Log and gives a detailed account of the meta-level features of this language. The main differences between ’Log and Prolog, as far as a WAM implementation is concerned, are discussed in the first part of chapter 4. The second part of this chapter is dedicated to describing the structural choices that have been made and the solutions that have been

adopted. Chapter 5 concludes this thesis with some evaluations and directions for future works.

It is assumed that the reader has knowledge of logic programming and, in particular, Prolog. The reader is refered to [StSh, Brat ,KlSz, Lloy87].

2. The Warren Abstract Machine.

2.1. Introduction.

The purpose of this chapter is to present the principal WAM concepts in sufficient detail so that the non-expert reader can understand the following discussion. For a deeper understanding, the reader is refered to [Warr, AïtK91a, AïtK91b].

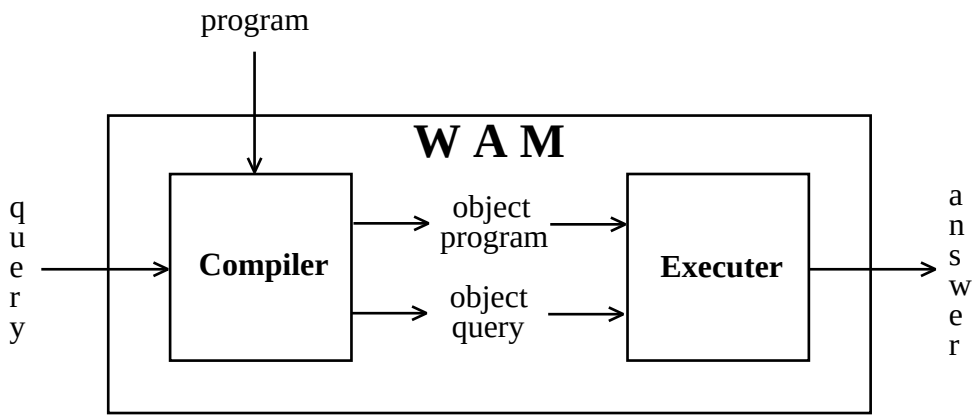

Figure 2.1 Architecture of the WAM.

A WAM system consists of two functionally distinct parts, as shown in figure 2.1. A source program is first processed by the **compiler**, that produces the WAM object code for that program, that is a sequence of assembly-like instructions. This object program is stored into the memory of the WAM and can eventually be saved into a file. When the user issues a query, it is first compiled. The resulting object query is combined with the current object program and given to the second module of the WAM, the **executer**. At this point, the actual execu-

tion starts and proceeds until the answer(s) to the query are produced (or a failure occurs).

Following Aït-Kaci's approach, the description below will be incremental: initially just unification will be considered, the resulting model will then be extended to multiple fact databases, then those single-clause procedures will be allowed to have a body and, finally, full Prolog will be implemented. The last section presents useful optimizations that can be easily included into the WAM design to obtain faster execution.

2.2. Basic unification.

Early Prolog implementations were interpreted because its computational model relies upon unification. Although unification is a highly dynamic process, large spaces are left for compilation. In fact, when unifying two terms, it usually happens that their syntactic structure is rich enough to be directly used in a compilation process. For example, when trying to unify the terms `f(a,b)` and `f(X,Y)`, the binary functor `f`, and the constants `a` and `b` are known in advance and can be compiled apart.

Consider initially the simple but fundamental problem of unifying two term; let's use Aït-Kaci's favorite example: `p(Z,h(Z,W),f(W))` and `p(f(X),h(Y,f(a)),Y)`. Call the former the *query* and the latter the *program.*

The basic idea of the WAM's treatment of unification is to *build the parse tree of the query and use the program to visit it*. Each time a node is visited, a match must be found with the corresponding object in the program, otherwise a failure occurs. This basic intuition is somewhat complicated by the fact that both the query and the program can contain variables. A variable in the pro-

gram is simply matched with whatever subterm is encountered in the query parse tree. A variable in the query is replaced with the parse tree of the subterm of the program encountered in its place. This is still quite simple but it is not yet complete, since a term can contain multiple occurrences of a variable. The idea is to use the first occurrence as a variable as its representation and consider the successive ones as pointers to it. This adds a new dynamic aspect when encountering a variable in the program: if it is not a first occurrence, the subterm on the parse tree and the subterm possibly bound to it by a previously encountered occurrence of the same variable in the program must be unified.

Let's schematize the executer's actions using Aït-Kaci's example and figure 2.2 as an illustration. Assume the parse tree has already been built. The executer is trying to match the program with the query parse tree. The following situations can occur (for the sake of conciseness, let's call p-node the current object in the program and q-node the current query node):

- both the p-node and the q-node are functors. They are checked for being the same functor. Figure 2.2(a);
- the q-node is a variable. The parse tree of the subterm associated to the p-node is built and all the occurrences of the variable are bound to it. Figure 2.2(c);
- the p-node is a first-seen variable. It is bound to the q-node. Figure 2.2(b);
- the p-node is a successive occurrence of a variable. The term previously bound to it is unified to the subterm associated to the q-node. Figure 2.2(d).

In order to make this discussion more concrete, three more items must be defined: the actual representation of what has been called the parse tree in the WAM's memory, the instruction set used by the executer, and the way the compiler transforms a Prolog term into a correct sequence of WAM instructions.

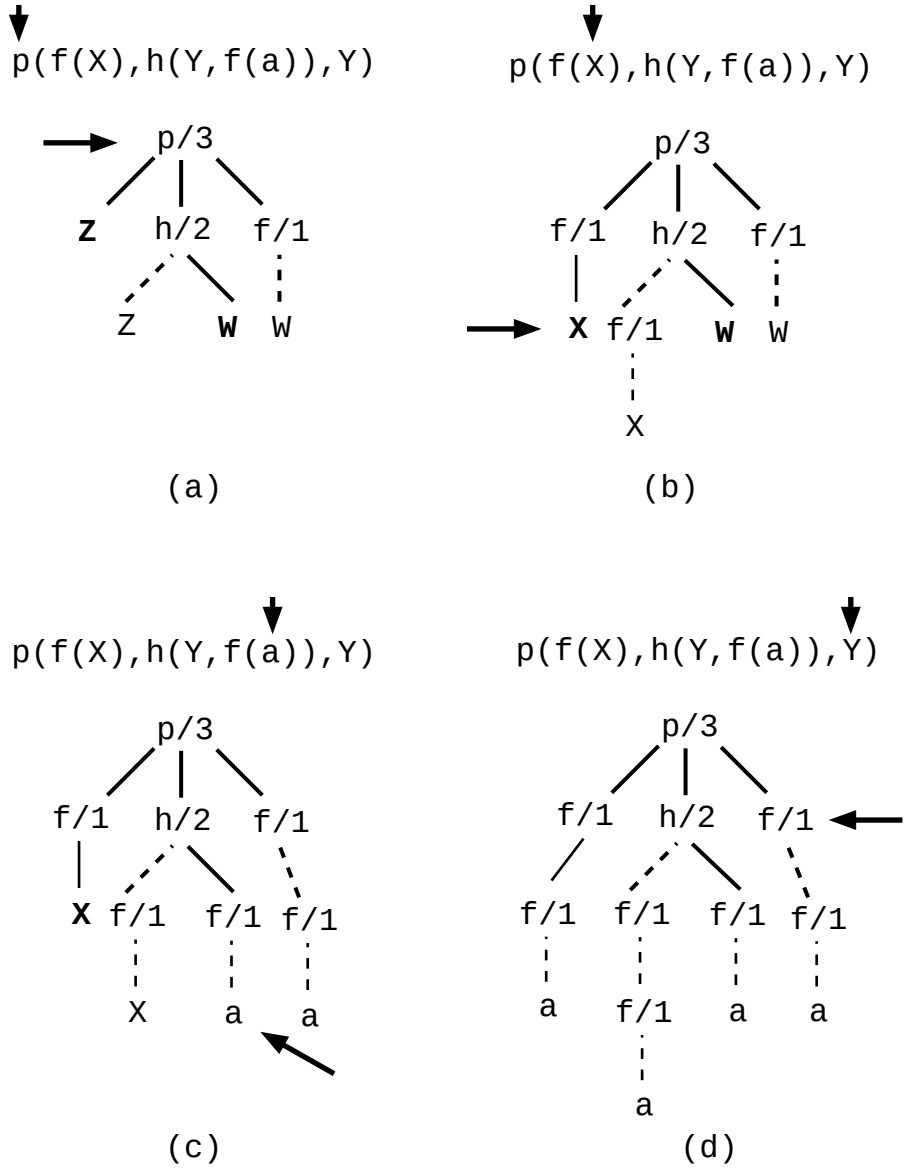

Figure 2.2 Steps of the unification algorithm.

(a) matching of a program functor with the corresponding query functor; (b) a first occurrence variable has been found in the query tree: it is substituted with the current program subterm; (c) the instantiation of a variable takes effect in all of its occurrences; (d) when a successive occurrence of a program variable is encountered, the term on the query parse tree must be unified with the term possibly bound to it.

2.2.1. WAM data representation.

The WAM stores Prolog terms into a memory area called the **heap**. It can be thought as an array of *data cells* managed in a LIFO fashion. A data cell is a fix-sized storage unit composed of the following fields: a *tag* identifying the cell type, a *flag* identifying subterms boundaries, and a type-dependent *value*. The following data cell types are needed (figure 2.3(a)):

- *functor cells*: These cells keep track of functor names. The data value field contains a functor identifier, for example a number randomly assigned with each functor.
- *structure cells*: These cells represent the link structure of the parse tree. Their value field is a pointer to a functor cell and consequently to a subterm of the currently represented term.
- *reference cells*: These cells represent variables. The value field of an unbound variable is a pointer to itself. Whenever the variable is bound, its value field is set to point to the subterm it has been bound to.

From a global point of view, a generic term $t = f(t_1,...,t_n)$ is represented as a sequence of n+1 consecutive data cells. The first is the functor cell of `f`, while each of the following ones is either a reference or a structure cell depending upon whether or not the corresponding subterm is a variable. If `t` is the outmost term, its structure cell is stored into a dedicated WAM register called the $\mathbf{X_0}$ **register**. Figure 2.3 (b) and (c) depicts the representation of the query term in the example both before and after the unification with the program term. They actually show a linearization of the parse trees of figure 2.2 (a) and (d). Therefore, the terms *heap structure* or *heap representation* will be used hereafter instead of *parse tree*. Note that since the arity of a term is not explicitly available, the flag field of the data cells is used to mark the last argument of a structured

term. Note also that the variables lose their name: heap addresses are used instead.

2.2.2. WAM instructions

Since a heap representation still represents a tree, building it requires the use of a series of *registers* to hold temporary data cells before they are pushed onto the heap. These registers are called X_0, X_1, X_2, ... X_0 has the task of pointing to the root of the tree.

Three instructions are needed to build the heap representation of the query.

- *put_structure f/n, X_i* pushes a functor cell for `f/n` into the heap and puts a structure cell pointing to it into the register X_i.
- *set_variable X_i* pushes a new reference cell onto the heap and copies it into the register X_i. This instruction is supposed to handle the first occurrence of a variable.
- *set_value X_i* copies the content of register X_i to a new cell on the top of the heap. This instruction handles already encountered variables as well as functor arguments.

Code for the query `p(Z,h(Z,W),f(W))` is shown in Figuree 2.4. One can easily check that it produces the heap configuration of figure 2.3(b).

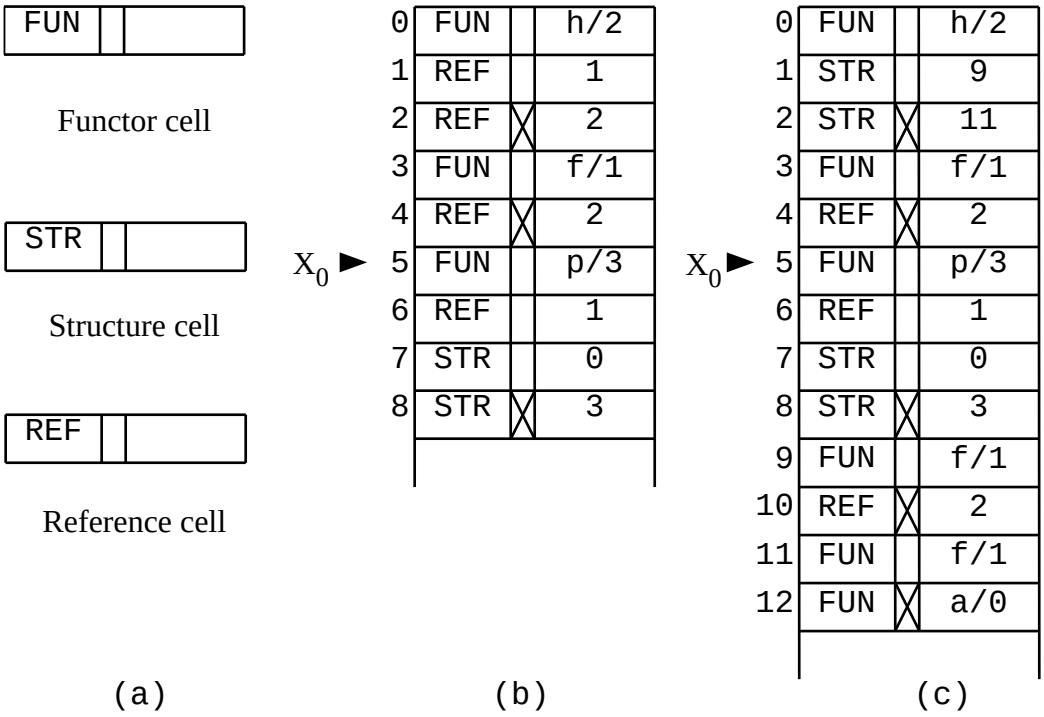

Figure 2.3 Data cells and term representation.

(a) The data cells. (b) Heap representation of `p(Z,h(Z,W),f(W))`. (c) Heap representation of `p(f(f(a)),h(f(f(a)),f(a)),f(f(a))`, after the execution.

```
put_structure   h/2, X2
set_variable    X1              % X1 <-> Z
set_variable    X4              % X4 <-> W
put_structure   f/1, X3
set_value       X4
put_structure   p/3, X0
set_value       X1
set_value       X2
set_value       X3
```

Figure 2.4 Query code for `:-p(Z,h(Z,W),f(W))`.

The program instructions are similar. Since they can be used both for matching objects as well as for building subterms, they can operate in two modes, called READ and WRITE. In READ mode, the address of the cell to be currently matched against is stored into an internal register called S.

- *get_structure f/n, X_i* checks if the heap cell pointed by register X_i is a functor cell for `f/n`. In this case it sets the mode to READ and S to the following cell. If X_i points to an unbound variable, it sets the mode to WRITE and pushes both a structure cell and a functor cell for `f/n` onto the heap.
- *unify_variable X_i*. In READ mode, it copies the cell pointed by S into X_i. In WRITE mode, it behaves as s*et_variable X_i*. S is incremented by 1.
- *unify_value X_i*. In READ mode, it unifies the term pointed to by S and the term pointed by X_i. In WRITE mode, it behaves as *set_value X_i*. S is incremented by 1.

```
get_structure   p/3, X0
unify_variable  X1
unify_variable  X2
unify_variable  X3                  % X3 <-> Y
get_structure   f/1, X1
unify_variable  X4                  % X4 <-> X
get_structure   h/2, X2
unify_value     X3
unify_variable  X5
get_structure   f/1, X5
unify_variable  X6
get_structure   a/0, X6
```

Figure 2.5 Program code for `p(f(X),h(Y,f(a)),Y).`

Code for the program `p(f(X),h(Y,f(a)),Y)` is shown in Figure 2.5. It can be easily checked that it produces the heap configuration of figure 2.3(c) once the code of figure 2.4 has been run.

2.2.3. Compilation of queries.

The next step is to show how to build a compiler to produce the code of figures 2.4 and 2.5 out of the corresponding Prolog terms. Aït-Kaci's method will be presented for simplicity, even if an actual implementation can act quite differently.

The first step when compiling a query is flattening it using temporary variables. The equation `X0 = p(Z,h(Z,W),f(W))` is in fact equivalent to a conjunction of equations having a maximum subterm nesting depth of 1; in this case:

```
X0 = p(X1,X2,X3)
X1 = Z
X2 = h(X1,X4)
X3 = f(X4)
X4 = W.
```

Now, arrange the equations in such a way that the first occurrence of a variable is always on the left-hand side. The equations involving only variables are then removed. This operation yields the following sequence:

```
~~X4 = W~~
X3 = f(X4)
~~X1 = Z~~
X2 = h(X1,X4)
X0 = p(X1,X2,X3).
```

Finally, write what remains on a single line substituting commas for the parentheses.

X3=f/1, X4, X2=h/2, X1, X4, X0=p/3, X1, X2, X3.

In this way, a sequence of tokens is obtained. Now, proceed the from left to right writing put_structure f/n, Xi when encountering Xi=f/n, set_variable Xi when encountering a variable Xi not previously seen and set_value Xi if Xi has already been encountered.

2.2.4. Compilation of programs

Compiling a program is similar. What changes is the order in which the equations must be rewritten during the second step. Since the data are already available on the heap and a matching must be performed, a variable must be read before it is checked. Therefore, after performing flattening and ordering, the program term `X0 = p(f(X),h(Y,f(a)),Y)` becomes:

X0 = p(X1,X2,X3)
X1 = f(X4)
X2 = h(X3,X5)
~~X3 = Y~~
~~X4 = X~~
X5 = f(X6)
X6 = a.

The linearization step produces:

X0=p/3, X1, X2, X3, X1=f/1, X4, X2=h/2, X3, X5, X5=f/1, X6, X6=a/0

Now, interpret each Xi=f/n as `get_structure f/n, Xi`, Xi either as `unify_variable Xi` or `unify_value Xi` depending upon whether Xi has already been encountered. That will produce the code of figure 2.5.

2.3. Facts.

Programs consisting of just one term are a good starting point for describing the WAM, but they are not very useful. Let's go a step further and admit as a program a sequence of Prolog facts, with the limitation that each of them must define a different predicate.

Once the query term has been built on the heap, there must be some way of selecting which program fact to use for the matching phase. Since the outer functors are unique in the program, the WAM approach is to use those functors to label the section of the program code for the corresponding facts. Once the query has been built on the heap, it is enough to jump to the label in the program corresponding to the outer functor of the query, or to fail if no fact is defined for this predicate.

Let's call the code area CODE and the register containing the address of the next instruction to be executed P. Two new instructions are needed. They are *control instructions* since they deal with the control flow of the program. They are:

- *call p/n* sets P to the address of the procedure defining p/n, if it exists.
- *proceed* terminates the execution.

Unfortunately, the idea of adding a *call* instruction at the end of the query code and a *proceed* after each compiled fact is too simplicistic: since the outer functors are not available, how can one identify the root of the trees? Warren's solution is to build a different parse tree for each argument of a predicate, either in the query or in the program. Let n be the arity of the current predicate, then the registers X_0, ..., X_{n-1} are reserved as pointers to the root of the heap representation of each of the predicate arguments. That means that when given the query $p(t_0, \ldots, t_{n-1})$, the executer must put the root of the representation of

t_i in Xi. If instead $p(t_0,...,t_{n-1})$ is a program fact, the executer expects the root of each t_i in Xi. Figure 2.6 illustrates the way queries and facts are compile.

Query		**Program fact**
$:-p(t_0,...,t_{n-1})$		$p(t_0,...,t_{n-1})$
<put t_0 in X_0>	p/n:	<get t_0 in X_0>
...		...
<put t_{n-1} in X_{n-1}>		<get t_{n-1} in X_{n-1}>
call p/n		proceed

Figure 2.6 Code pattern for a query and a program fact.

Because of their specialized use, when compiling a n-ary predicate, the registers $X_0,...,X_{n-1}$ have been traditionally renamed $A_0,...,A_{n-1}$, the *A* stands for *Argument*. This is just an alias: they are not different registers.

Unfortunately, there is still a problem. The arguments of a predicate could be variables. Four new instructions are defined to handle this case, two for the queries and two for the program facts; two for first the occurrence of a variable and two for the successive occurrences:

- *put_variable* X_n, A_i: This instruction is used when compiling a first-seen occurrence of an argument variable in a query. It pushes an unbound reference cell on the heap and copies it both in X_n and in A_i. A_i will be used by the program while X_n represents the variable for an eventual successive use in the query.
- *put_value* X_n, A_i: This instruction handles already seen variable occurrings as arguments in a query. It simply copies the content of X_n into A_i.

- *get_variable* X_n, A_i: This instruction is used when compiling a first-seen occurrence of an argument variable in a program fact. It copies the content of A_i into X_n, the official reference to the variable.
- *get_value* X_n, A_i: This instruction handles the case of a successive occurrence of a variable in a program fact. It unifies the contents of registers X_n and A_i.

Figures 2.7 and 2.8 show the result of the compilation of the query and the fact of the previous section.

```
put_variable      X3, A0        % X3 <-> Z
put_structure     h/2, A1
set_value         X3
set_variable      X4            % X4 <-> W
put_structure     f/1, A2
set_value         X4
call              p/3_0
```

Figure 2.7 Code for `:-p(Z,h(Z,W),f(W)).`

```
p/3: get_structure     f/1, A0
     unify_variable    X3            % X3 <-> X
     get_structure     h/2, A1
     unify_variable    X4            % X4 <-> Y
     unify_variable    X5
     get_structure     f/1, X5
     unify_variable    X6
     get_structure     a/0, X6
     get_value         X4, A2        % X4  <-> Y
     proceed
```

Figure 2.8 New program code for `p(f(X),h(Y,f(a)),Y).`

2.4. Flat resolution.

It is necessary for program clauses to have a multi-atom body. Queries too may be composed of more than one atom. Nevertheless, the same limitation as in the previous section still applies: each predicate in the program must be defined by at most one clause.

According to standard Prolog resolution, once the head of a clause has been successfully unified with some atom in the current query (the leftmost actually), its body becomes (part of) the query. The idea is to compile the head of a rule as a fact in the previous section and its body as a query, i.e. $h{:-}b_1,...,b_n$ is compiled (informally) as:

h: <*get* code for h>
<*put* code for b_1>
`call` b_1
...
<*put* code for b_n>
`call` b_n
`proceed.`

Some refinements are needed to make this approach work. There are two problems. First after the completion of the execution of a procedure, the control flow must be resumed at the instruction following the corresponding `call`. Second the scope of variables is not local to the single atoms, as in the previous sections, but is spread over the whole clause. Care must be taken so that after a `call`, the value of previously encountered variables is still available.

Let's first focus on the control problem. Initially, consider the case of the facts. The problem can be easily solved with a single register, called the *Continuation Point register*, or *CP*. Whenever a `call` is performed, the address of the following instruction (i.e. P+1) is saved in CP. Now, it is enough to modify

`proceed` by having it set P to the content of CP, i.e. to the instruction following the `call` to the fact..

This solution is still too weak for rules. In fact, a rule can call another rule that can call a third rule and so on. One must keep track of the succession of return points of all the active rules. The same problem exists with procedural programming languages [AhSU] and the solution adopted in the WAM architecture is similar: the use of a control *stack*. The control stack is called STACK and the address of the current environment is called E. Each time a rule is called, it should allocate a new frame (called an *environment*) on the top of the stack to record the current continuation point (the value of CP). When a rule terminates its execution, it recovers the continuation point from the topmost environment on the stack, and pops that environment. So, two new (control) instructions are needed, to be executed upon entering and leaving a rule, respectively:

- *`allocate`*: This instruction allocates an environment on the top of the stack and consequently modifies E. The environment contains the current content of the register CP (set by call) and the previous value of E (for deallocating the frame).
- *`deallocate`*: This instruction recovers the value of CP and deallocates the current environment. Since it would always followed by proceed, the two effects can be combined: instead of loading the continuation point in CP and then executing proceed, deallocates directly loads the continuation point in P.

Schematically, rules and facts are compiled in the following way (informally):

Fact	Rule
`h.`	`h :-` b_1`,...,`b_n`.`
<*get* code for h> `proceed`	h: `allocate` <*get* code for h> <*put* code for b_1> `call` b_1 ... <*put* code for b_n> `call` b_n `deallocate`

Figure 2.9 Control-correct code pattern for facts and rules.

The last problem to solve is how to manage the value of the variables whose scope extends across the rule. Consider for example the rule

p(X,Z) :- q(X,Y), r(Y,Z)

which is compiled as:

```
p/2: allocate
     get_variable    X2, A0      % X <-> X2
     get_variable    X3, A1      % Z <-> X3
     put_value       X2, A0
     put_variable    X4, A1      % Y <-> X4
     call            q/2
     put_value       X4, A0
     put_value       X3, A1
     call            r/2
     deallocate
```

After `q/2` has been called, there is no guarantee that `X3` and `X4` still contain the values of `Y` and `Z` respectively since those registers could have been used by the clause defining `q/2`. Note that there are no problems for `X2`.

Aït-Kaci classifies the variables appearing inside a Prolog clause into two types:

- the *permanent variables* are those variables appearing in at least two goals in the body of the clause. The head is considered part of the first body atom.
- the *temporary variables* are all the other variables, that are local in a sense to a body goal.

In the previous example, `Y` and `Z` (i.e. `X3` and `X4`) were permanent while `X` (i.e. `X2`) was temporary.

The permanent variables are exactly those having two occurrences in the WAM code separated by a call instruction. Some actions must be taken to have them outlive the procedure calls. Note that temporary variables do not cause any problem. The WAM records permanent variables into the environment frame associated with the rule in which they appear, instead of using a register. In this way, any access to a permanent variable after a call will find the value it had before the call on the stack. Note that this value could have been updated by the procedure call, as the result of a unification for instance, but is not lost. The variable on the stack, as well as the registers, are in fact pointers to the heap.

The permanent variables of a clause are denoted by $Y_0, \ldots, Y_{n-1}$. Note that the previously defined instructions must be updated to accept permanent variables. In order to accommodate the permanent variables in the environment frame, the format of `allocate` must be modified:

- *`allocate N`* allocates an environment frame for *N* permanent variables on the top of the stack and modifies E.

The format of an environment frame is depicted in figure 2.10

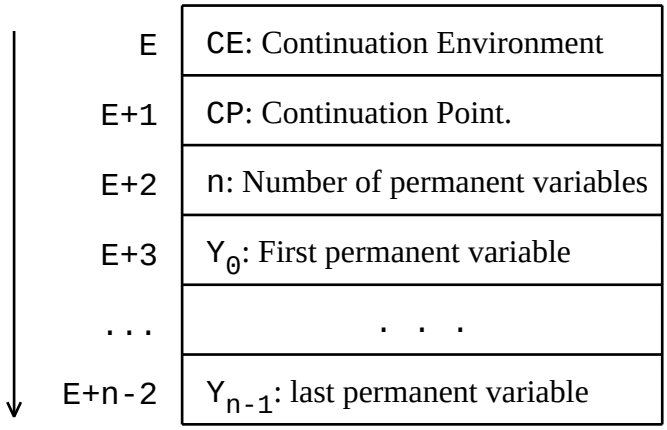

Figure 2.10 Environment frame.

Fact	**Rule**
`h.`	`h :- b1,...,bn.`

<*get* code for h>
`proceed`
	`h:` `allocate N`
	<*get* code for h>
	<*put* code for b_1>
	`call` b_1
	...
	<*put* code for b_n>
	`call` b_n
	`deallocate`

Figure 2.11 Code pattern for facts and rules.

```
p/2: allocate       2
     get_variable   X2, A0      % X <-> X2
     get_variable   Y0, A1      % Z <-> Y0
     put_value      X2, A0
     put_variable   Y1, A1      % Y <-> Y1
     call           q/2
     put_value      Y1, A0
     put_value      Y0, A1
     call           r/2
     deallocate
```

Figure 2.12 WAM code for `p(X,Z):-r(X,Y),q(Y,Z).`

Figures 2.11 and 2.12 show, respectively, the definitive code pattern for facts and rules (and consequently queries) with *N* permanent variables and the correct compilation of the clause `p(X,Z):-r(X,Y),q(Y,Z).`

2.5. Pure Prolog.

In order to obtain pure Prolog from the language defined in the previous section, it is enough to allow multiple clause definitions for the predicates in a program. Modifying the executer to handle this new degree of freedom is not completely straightforward because a unification failure is no longer definitive; alternative clauses could be tried. Backtracking on an alternative clause C' of a procedure P requires the executer to return to the exact situation that was in effect before the current clause C of P was called. In particular, all the actions made by the partial execution of C must be undone.

Let $C_1,...,C_n$ be the clauses defining procedure P. Backtracking requires the following informations to be saved either before running C_1 or during the execution of any C_i:

- the argument registers set by the clause calling P, since they will not be available after backtracking;
- a list of all the variable bindings performed after entering P, since they must be undone when backtracking occurs;
- the top of the heap when entering C_1, since all the heap space used afterwards can be recycled after backtracking;
- the next clause to try (C_{i+1}) in the case the current clause (C_i) fails;
- pointers to the environment and information record of the previous multi-definition clause, since multi-definition procedures can be nested and all the Ci's can fail.

All the above values are collected into a record called a *choice point*. Choice points can be conveniently put on the environment stack [AïtK91b]. The stack structure is used because a multi-definition procedure can call another one that in turn can call a third, etc. The environment stack protects environments that were active at the moment of the choice point creation and that have since been discarded, but that must be recovered when backtracking. The stack address of the current choice point is always available in an internal register called B.

The actual contents of a choice point are illustrated in figure 2.13. Each time an unbound variable is bound to an object, the heap address of the variable is recorded on a special purpose stack, called the *trail*. The top of the trail is held in an internal register called TR. When a multi-definition procedure is entered, that is when its choice point is allocated on the stack, the current top of trail is recorded in the choice point. In this way, all the successive bindings will be recorded on the trail above that point. The actions to be taken upon backtracking are now straightforward: reset to unbound all the heap addresses present

on the trail between the current top of trail and the value TR had when entering the procedure (and reset TR to that value).

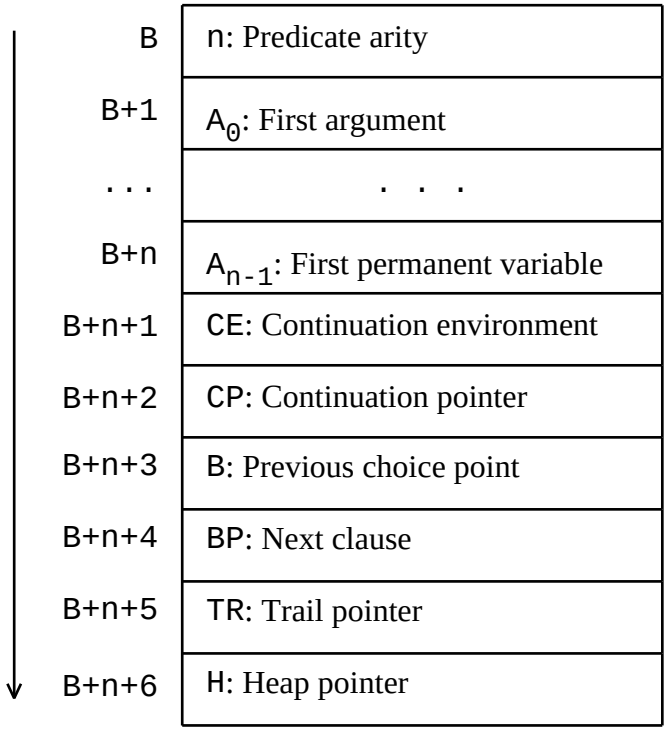

Figure 2.13 Choice point.

Choice point handling requires two new WAM instructions to create and maintain them:

- *try_me_else L, N*: this instruction allocates a choice point on the top of the stack. L is the label of the next clause of that procedure to try and N is arity of the head predicate, i.e. the number of arguments. The argument registers A_0, ..., A_{N-1} are copied into the choice point;
- *retry_me_else L*: this instruction updates the current choice point after backtracking has occurred. It performs all the recovery actions previously de-

scribed, i.e. it undoes the bindings made by the failed call, it resets the top of heap and it loads the procedure calling arguments back into the argument registers. L is the label of the next clause of that procedure to try, if there are any, otherwise it is the special label FAIL. In the latter case, the choice point is deallocated and one backtracking step is performed: the next choice point on the stack is considered. Otherwise, L is set as the label of the next clause to try, in case of failure of the current trial.

Compiling a procedure P consisting of the clauses C_0,...,C_n is a simple generalization of what has been seen in the previous section. If the procedure consists of a single clause (i.e. n = 0), nothing more is needed than the compiled code of C_0. Otherwise, before entering C_0, a choice point must be allocated. This choice point must be updated before entering all the successive C_i's but the last, when it should be discarded. Figure 2.14 illustrates more precisely the outcome of a procedure compilation.

Single definition procedure

```
                C0
L0:     <code for C0>
```

Multi-definition procedure

```
                C0,...,Cn+1
L0:     try_me_else L1,N
        <code for C0>
L1:     retry_me_else L2
        <code for C1>
        ...
Ln:     retry_me_else Ln+1
        <code for Cn>
Ln+1:   retry_me_else FAIL
        <code for Cn+1>
```

Figure 2.14 Code pattern for procedure definitions.

Figure 2.15 illustrates compilation process by showing the way the classical definition for append/3 is compiled. The low-level `./2` list constructor has been used. The resulting code is shown in figure 2.15.

```
                    append([],L,L).
                    append(.(A,L1),L2,.(A,L3)) :-
                                        append(L1,L2,L3).

append/3_0:  try_me_else     append/3_1, 3
             get_structure   []/0, A0
             get_variable    X3, A1          % L <-> X3
             get_value       X3, A2
             proceed.
append/3_1:  retry_me_else   FAIL
             allocate        0
             get_structure   ./2, A0
             unify_variable  X3              % A <-> X3
             unify_variable  X4              % L1 <-> X4
             get_variable    X5, A1          % L2 <-> X5
             get_structure   ./2, A2
             unify_value     X3
             unify_variable  X6              % L3 <-> X3
             put_value       X4, A0
             put_value       X5, A1
             put_value       X6, A2
             call            append/3_0
             deallocate
```

Figure 2.15 WAM code for append/3.

2.6. Optimizations.

Even though the design presented so far is complete, in the sense that the instruction set and the internal data structures suffice to compile and execute

a Prolog program, a number of optimizations are commonly employed. This section presents a few such optimizations that are very easy to implement and produce substantial improvements in the execution speed. These and other optimizations can be found in [AïtK91b].

As far as the WAM is concerned, efficiency can be improved in several ways. Substantial heap space can be saved defining special purpose instructions for very common situations, like dealing with constants and lists. This has the effect of collapsing several WAM instruction in a single specialized instruction, reducing the code size of a compiled program. Also recycling registers that will not be used in the future can cut the number of registers used by several orders of magnitude. In addition, this has the side-effect of pointing out useless instructions that can be eliminated from the code. Another target of the optimization is the stack: the idea is to use at each instant only the part of an environment that is really needed.

2.6.1. Constants.

Let's observe the way constants (i.e. nullary functors) are compiled and represented on the heap. As an example consider the subterm `f(b,g(a))`. Figure 2.16 (a), (b) and (c) represent the code for that subterm when it occurs in the head and in the body of a clause, and the heap term constructed by the body code respectively. Note that each occurrence of a constant in the source code is compiled as two WAM instructions and that each constant is represented on the heap using two data cells. Constants are very common objects in a Prolog program, so a new type of data cell has been introduced for them: *constant cells*. A constant cell is intended to collapse the link of the parse tree leading to the constant and the constant name into one heap entity. Its tag is CON and its value field contains the constant itself. Handling constant cells requires the introduction

of four new WAM instructions, two for the head of a clause and two to be used when compiling the body, two for constants occurring as arguments and two for inner occurrences of constants:

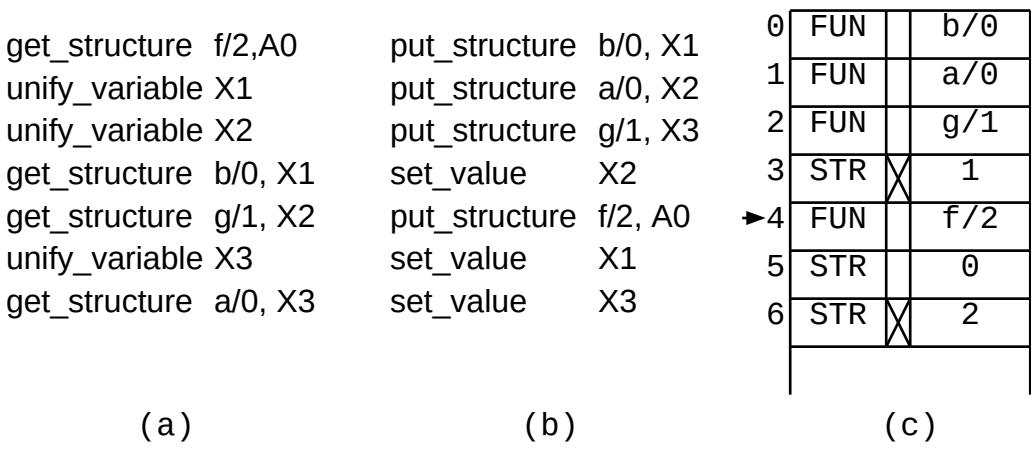

Figure 2.16 Compilation and heap representation of `f(b(g(a))`.

- *`put_constant c, A`$_i$*: is used when compiling a constant occurring as an argument of the body of a clause. It simply loads A_i with a constant cell for `c`;
- *`get_constant c, A`$_i$*: is used for constants occurring as arguments in the head. If A_i points to a constant, it is checked for being `c` itself. If A_i points to an unbound reference cell, it is set to a constant cell for `c`;
- *`set_constant c`*: used when encountering an inner constant in the body of a clause, this instruction pushes a constant cell for `c` on the top of the heap;
- *`unify_constant c`*: in READ mode, it checks if the currently examined cell on the heap is a constant cell for `c;` if it is instead an unbound reference cell, it binds it to a constant cell for `c`. In WRITE mode, it behaves as set_constant.

Figure 2.17 illustrates the result of the compilation using these new instructions. Notice how both the code length and the heap representation length have been substantially reduced.

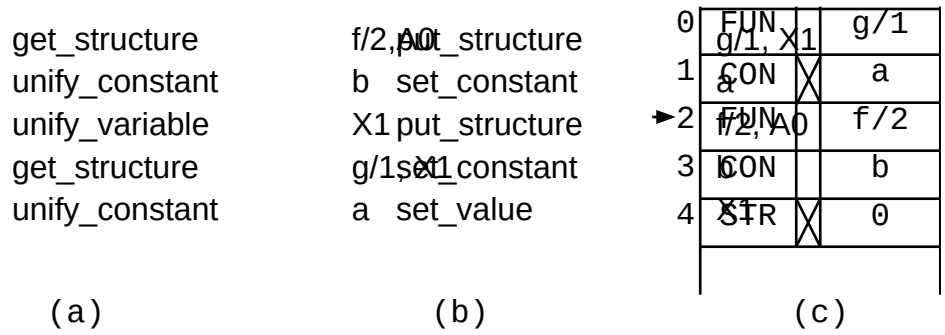

Figure 2.17 Enhanced compilation and heap representation of `f(b(g(a))`.

2.6.2. Lists

Lists are a very commonly used data structure in Prolog programming. Therefore it makes sense to devise specialized instructions for them. The idea is to introduce a new type of data cell, called a *list cell*, that collapses the link to a list subterm (i.e. its structure cell) and the functor cell for the list constructor. The tag of a list cell is LIS and its value field points to the data cell representing the first argument of the now implicit list constructor `./2`. Two instructions can be added to the WAM instruction set to handle this new heap representation for lists:

- *`put_list`* X_i substitutes the instruction `put_structure ./2,` X_i. It inserts in X_i a list cell pointing to the next available heap location.
- *`get_list`* X_i substitutes the instruction `get_structure ./2,` X_i. It checks if the location pointed by X_i is a list cell and in this case, the next heap cell to visit is set to the cell pointed by it. If instead it is an unbound reference

cell, it is bound to a new list cell pointing to the next available location on the heap and the mode is set to WRITE.

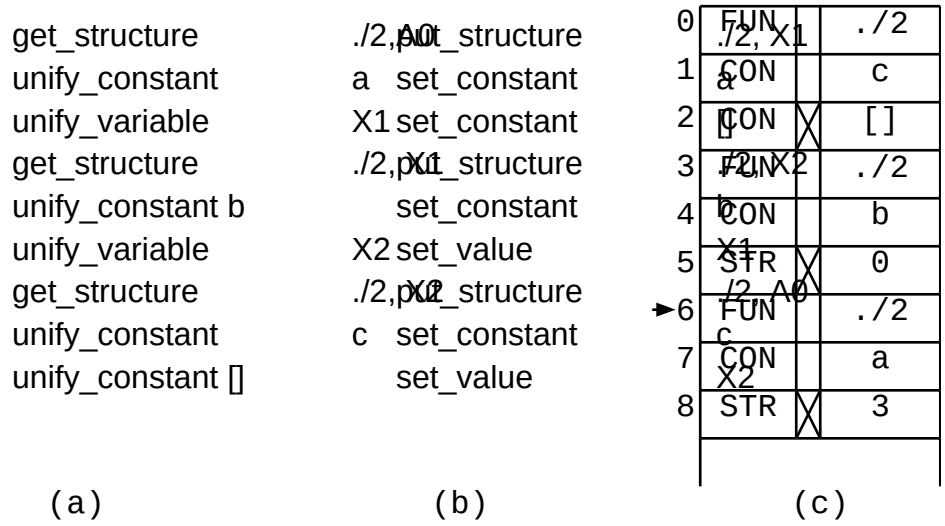

Figure 2.18 Compilation and heap representation of [a,b,c].

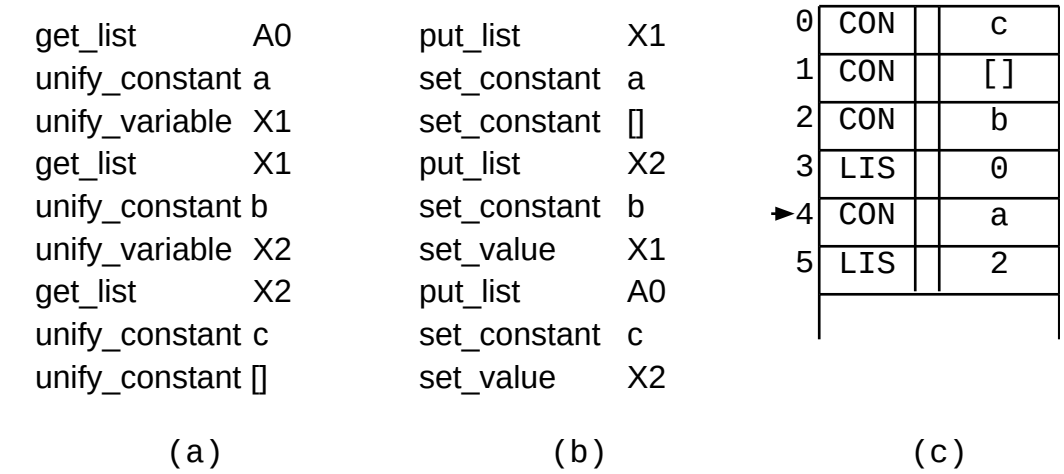

Figure 2.19 Enhanced compilation and heap representation of [a,b,c].

Figures 2.18 and 2.19 shows respectively code and heap representation for the term `[a,b,c]` both before and after the enhancement just presented. Note that the heap representation has been reduced in size.

2.6.3. Registers

In the WAM code presented so far, no attempt has been made to recycle registers that will not be used later. A clever register allocation policy can save hundred of registers. For example, a first version of the compiler run on an example program (the wolf-goat-cabbage problem) used 89 registers, while this number was reduced to 5 after improving the allocation policy.

An even more clever register allocation policy can reduce the number of code instructions. For instance, the combined effect of the couple of instructions: `get_variable X3, A0` followed by `put_value X3, A0` is null. So these two instructions can be eliminated. Figure 2.20 shows how the code for append/3 can be reduced in size by careful register allocation.

2.6.4. Miscellaneous stack optimizations.

Stack space is a resource that should be used as sparingly as possible. Consider a non fact rule. Its last two instructions are always a `call` to some procedure followed by `deallocate`. Once all the argument registers have been properly loaded, the only reason for maintaining the current environment on the top of the stack is for the continuation point. The idea is therefore to swap the `call` and `deallocate` instructions. Unfortunately, this entails the introduction of a new instruction, *`execute p/n`* that behaves like `call p/n` but does not set the continuation pointer CP to the next instruction (that would be erroneous). This technique is known as *last call optimization*.

```
append/3_0:
    try_me_else     append/3_1, 3
    get_constant    [], A0
    get_variable    X3, A1
    get_value       X3, A2
    proceed.
append/3_1:
    retry_me_else   FAIL
    allocate        0
    get_list        A0
    unify_variable  X3
    unify_variable  X4
    get_variable    X5, A1
    get_list        A2
    unify_value     X3
    unify_variable  X6
    put_value       X4, A0
    put_value       X5, A1
    put_value       X6, A2
    call            append/3_0
    deallocate
```

(a)

```
append/3_0:
    try_me_else     append/3_1, 3
    get_constant    [], A0
    get_value       A1, A2
    proceed.
append/3_1:
    retry_me_else   FAIL
    allocate        0
    get_list        A0
    unify_variable  X3
    unify_variable  X4
    get_list        A2
    unify_value     X3
    unify_variable  X5
    put_value       X4, A0
    put_value       X5, A2
    call            append/3_0
    deallocate
```

(b)

Figure 2.20 Code for `append/3` before and after improving register allocation.

Similarly, after each call in the body of a rule, the number of permanent variables that are still needed decreases. Therefore, a clever ordering of the permanent variables in the environment can save stack space. After each call, a permanent variable that will not be used later can be easily discarded if it occupies the topmost positions in the environment. This space can be used by successive procedures. The only change that needs to be performed is to provide a new

argument to `call`. The new format of this instruction is *`call p/n, N`*, where N is the number of permanent variables that will be used later. This technique is called *environment trimming*.

A further optimization can be done for rules having a single atom in their body, called *chain rules*. For these rules, there is no need to allocate an environment since they do not have permanent variables and the continuation pointer is handled by `execute`. Figure 2.21 shows how the compiled code for append/3 looks like after applying these above optimizations:

```
append/3_0:
        try_me_else     append/3_1, 3
        get_constant    [], A0
        get_value       A1, A2
        proceed.
append/3_1:
        retry_me_else   FAIL
        get_list        A0
        unify_variable  X3
        unify_variable  X4
        get_list        A2
        unify_value     X3
        unify_variable  X5
        put_value       X4, A0
        put_value       X5, A2
        execute         append/3_0
```

Figure 2.21 Definitive WAM code for `append/3`.

Further optimization techniques beyond the scope of this thesis can be found in [AïtK91b, Warr].

2.7. Summary.

Let's now summarize the WAM concepts introduced in this chapter. Figure 2.22 shows the instruction set of the WAM, organized according to the use and the type of instructions. Figure 2.23 illustrates the internal data structures utilized by the WAM.

Head		**Body**	
	Argument instructions		
get_variable	X_n/Y_n, A_i	put_variable	X_n/Y_n, A_i
get_value	X_n/Y_n, A_i	put_value	X_n/Y_n, A_i
get_structure	f/n, X_i	put_structure	f/n, X_i
get_constant	c, A_i	put_constant	c, A_i
get_list	X_i	put_list	X_i
	Term instructions		
unify_variable	X_n/Y_n	set_variable	X_n/Y_n
unify_value	X_n/Y_n	set_value	X_n/Y_n
unify_constant	c	set_constant	c
unify_void	n	set_void	n
	Control instructions		
allocate	N	deallocate	
		call	p/n, N
		execute	p/n
		proceed	
	Choice point instructions		
try_me_else	L, N		
retry_me_else	L/*FAIL*		

Figure 2.22 The WAM instruction set.

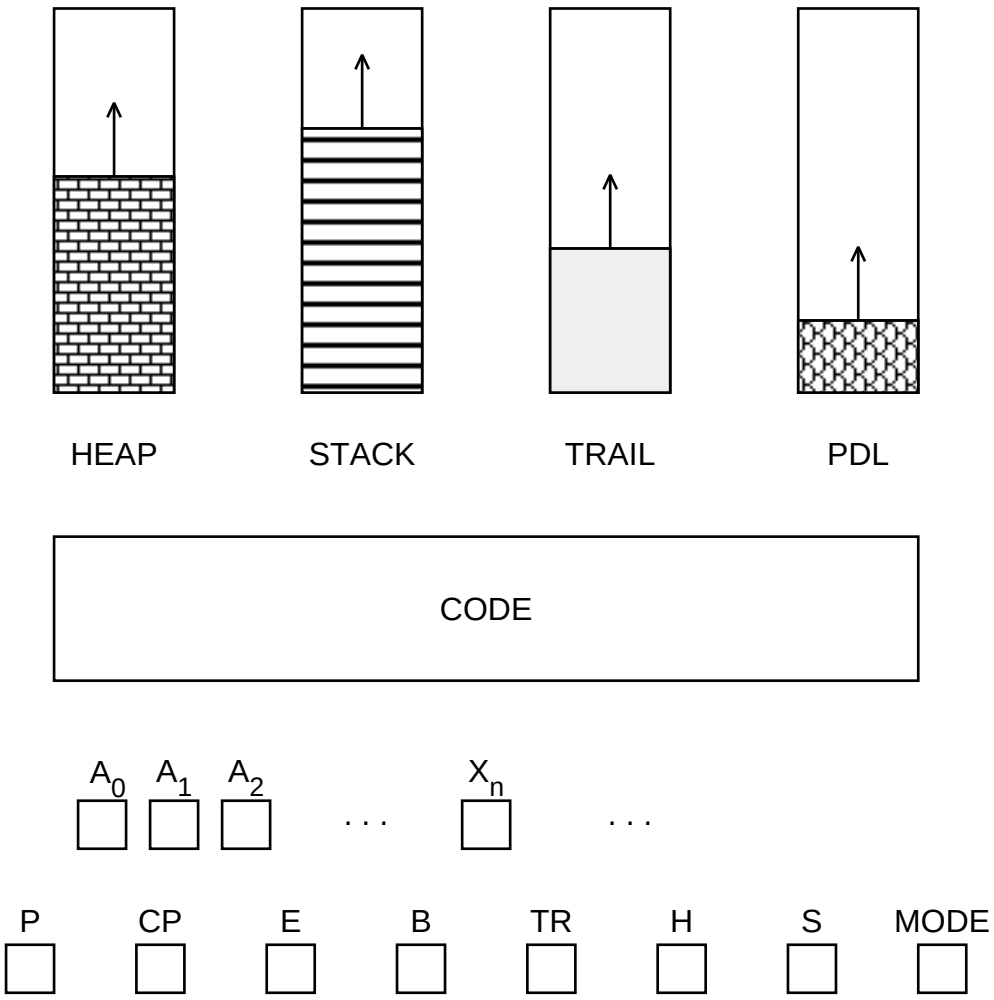

Figure 2.23 The WAM working areas and registers.

3. Logic meta-programming in ’Log.

3.1. Introduction.

Because meta-programming languages are supposed to operate on object programs as data, a key feature for these languages is their ability to to *represent* and *manipulate* the object level entities. The representation issue is achieved by means of a *naming schema*: each entity of the object level is given a unique *name* used at the meta-level as the (only) way to operate on the object entity it denotes. Therefore, a meta-programming does not directly deal with programs; it instead manipulates the names of these programs, that are isomorphic to the programs themselves.

A good introduction to meta-programming in logic programming can be found in [BoKo]. In this context, the most interesting case is when the object language and the meta-language coincide. In this case, the naming scheme is a form of *autorepresentation*. When the meta-language is a logic programming language, the name of any object entity is a meta-level term. Note also that since programs are composite objects, a meta-logic programming language should provide a way of naming not only programs, but also their components, such as clauses, terms and symbols (an alternative is to introduce predefined predicates to perform fixed operations on program names).

’Log has been developed along these lines to fulfilling the following requirements:

- to treat at the meta-level **any** syntactic entity of the language, from the symbols up to the programs;
- to allow an **efficient implementation**, in terms of both run-time overhead and memory usage;
- to possess a sound and complete **logical semantics** for the whole language, and in particular for the meta-level features;
- to offer the users simple, powerful and **easy-to-use** meta-programming facilities.

These goals have emerged from observing the deficiencies of the meta-logic programming languages proposed in the decade following Bowen and Kowalski's analysis. Some of the most important contributions in that field are: MetaProlog [Bowe, BoWe], λ-Prolog [MiNa86, MiNa87], Reflective Prolog [Cost, CoDL, CoLa, Dell] and Gödel [HiLl91].

'Log is a pure logic programming language [Lloy87] augmented with facilities designed to permit effective meta-programming facilities, in a logical framework. What distinguishes this language from its predecessors is the naming schema used. Each 'Log syntactic entity has, in fact, two meta-representations associated with it, i.e. two ground terms that uniquely describe it and allow it to be manipulated at the meta-level.

The first meta-representation associated with a syntactic expression is a constant symbol - called its **name** - which is isomorphic in structure to the expression it represents. Objects having a name in 'Log are programs, clauses, terms, symbols and characters. Giving characters a name might be surprising at first: in fact, a character is not a proper syntactic expression in a logic language. However, the ability to access characters, even via their name, appears to be quite

useful in practice. It allows for example a semantically acceptable definition of the Prolog built-in *var/1*.

Each proper syntactic entity (i.e. not characters) has, beside its name, a **structural representation**, i.e. a ground term that describes the structure of the entity in terms of the names of its components. The name and the structural representation are two descriptions of a syntactic object at two different levels of detail.

'Log names are syntactic entities; therefore they have a name and a structural representation too. Thus, 'Log supports the definition of an infinite tower of meta-levels. In most applications, only an object level and a single meta-level are actually needed.

For each composite syntactic object, it is possible to pass from its name to its structural representation and vice versa by means of a series of new operator <=*x*=>/2, where x depends on the type of object being transformed.

'Log does not supply any built-in reflection operator which would allow a meta-representation to be obtained from the object it denotes or vice versa. Neither it is possible to define such an operator using 'Log. This differentiates 'Log from other proposals, such as Reflective Prolog and R-Prolog* [Suga], that assume a reflection mechanism to be available. In 'Log meta-levels are strictly separated: at each level, the syntactic entities of the lower levels are visible through their names only; variables do not make an exception to this rule.

Names, structural representations and <=x=> suffice to define (declaratively) every Prolog meta-predicate. Thus it is possible to include these predicates in a 'Log program without compromising its logical interpretation.

The next section describes the way names are defined in ’Log as well as the key properties of these objects. Section 3.3 focuses instead on structural representations. The following section describes the bridge existing between the two meta-representation of each ’Log’s syntactical entity. Section 3.5 shortly presents the main theoretical results valid for this language. Finally, section 3.6 describes features of ’Log that have not been taken into consideration previously.

3.2. Names.

The syntax of ’Log is similar to that of Prolog and will be skipped here, except for those parts concerning the meta-representations. Unlike the usual syntactic definition of a logic program, ’Log programs are not built out of a set of functional, predicate and variable symbols. Their basic components are instead **characters** belonging to an **alphabet** Σ. The main use of Σ is in the construction of the language symbols. A symbol can thus be decomposed into its constituent characters, and have them treated as ’Log syntactic entities. Σ is subdivided into an upper case alphabet and a lower case alphabet. In this way it is possible to adhere, as far as possible, to the syntactic conventions of standard Prolog.

There are three different kinds of **symbols** in ’Log. Beside the usual functional symbols and variables, ’Log introduces names as a third basic syntactic class. In order to provide a definition of names, the notions of program, clause and term need to be introduced first.

Terms are defined in ’Log in the usual recursive way on symbols. In particular, since a name is a symbol, a name is a term too. ’Log **clauses** differ from standard Horn clauses because atomic formulas are terms. In fact, there is no distinction between predicate and functional symbols in ’Log. Finally, a ’Log

program is a sequence of clauses separated by dots. In 'Log, the notion of sequence is more appropriate than the usual notion of set.

The definition of **name** can now be given as follows:

- if P is a program, then pg(P) is the **program name** of P;
- if C is a clause, then cl(C) is the c**lause name** of C;
- if t is a term, then tr(t) is the **term name** of t;
- if s is a symbol, then sy(s) is the **symbol name** of s;
- if c is a character, then ch(c) is the **character name** of c.

Each symbol has, as in Prolog, an associated arity (equal to 0 for variables and names). Notice that even if the syntax is somewhat misleading, names are atomic objects, in the same way as constants are. Therefore, *pg*, *cl*, *tr*, *sy* and *ch* are not to be considered in this framework as functors, but as constant constructors. In fact, their purpose is only to maintain a visible relationship between an object and its name. Furthermore, since names are symbols it is evident that the two term names `tr(p(X))` and `tr(p(Y))` do not unify, whereas the variable `X` unifies with the term `tr(f(X))`, yielding the substitution {`X` $\rightarrow$ `tr(f(X))`}, without any prevention from the occur check.

Notice that the same syntactic expression can be denoted by different names, depending on the context where occurs. Consider the string containing just the character `c`. According to 'Log's syntax, `c` can be either a character or a symbol or a term. Thus, the character name `ch(c)`, the symbol name `sy(c)` and the term name `tr(c)` are all meta-representations for this same string

As an example, consider the usual `append/3` predicate, which concatenates two lists. Its program name is:

```
pg(
    append([],X,X):-.
    append([A|X],Y,[A|Z]):-append(X,Y,Z).
    )
```

where the list notation has the usual definition.

As names are symbols, they have a name too. Thus, for instance, the term names of the character name `ch(c)`, of the symbol name `sy(foo)` and of the term name `tr(Alpha)` are `tr(ch(c))`, `tr(sy(foo))` and `tr(tr(Alpha))` respectively. 'Log does not impose any limit on the number of naming levels.

3.3. Structural representations.

As already mentioned in section 3.1, 'Log associates to every composite syntactic object (i.e. symbols, terms, clauses and programs) a second meta-representation, called the **structural representation** of the object. This meta-representation is a ground term which describes the structure of the object it denotes in terms of the names of its components.

The structural meta-representation of an object usually takes the form of a list of name constants. Terms are an exception: they require embedded lists of symbol names to account for their recursive definition.

A precise definition of the structural representation of the various 'Log syntactic entities is now given:

- if $P = C_1.C_2. \ldots C_n.$ is a program, then
  $[cl(C_1),cl(C_2), \ldots ,cl(Cn)]$
  is the **program structure** of P; the program structure of the empty program is represented by the empty list;

- if C = A :- A_1,A_2, ... ,A_n is a clause then

  [tr(A),tr(A_1),tr(A_2), ... ,tr(A_n)]

  is the **clause structure** of C, if C is a query, the first element of the list is [];

- the **term structure** t' of the term t = f(t_1,t_2, ... ,t_n) where n > 0 is recursively defined as

  [sy(f),t_1',t_2', ... ,t_n']

  if t = s is a constant, a variable or a name, its term structure is

  sy(s);

- if s = $c_1c_2...c_n$ is a symbol, then

  [ch(c_1),ch(c_2), ... ,ch(c_n)]

  is the **symbol structure** of s.

While names are atomic symbols, structural representations are compound ground terms. In this framework, terms similar to structural representations apart from the occurrence of meta-level variables inside them, must be considered as **partially specified** structural representations. For instance,

```
[sy(f),X]
```

is not a term structure; however, if the (meta-level) variable `X` is instantiated either to the symbol name `sy(a)`, or to the term structure `[sy(g),sy(a),sy(b)]`, the correct structural representations `[sy(f),sy(a)]` and `[sy(f),[sy(g),sy(a),sy(b)]]` are respectively obtained. Meta-level variables occurrences in an incomplete structural representation will be called **meta-variables**, in contrast with the object variables that are frozen inside the names that constitute the structural representation.

In contrast to names, structural meta-representations do obey the law of unification: `[sy(f),X]` and `[sy(f),sy(Alpha)]` unifies, yielding the

substitution {X → sy(Alpha)}, whereas the goal X=[sy(f),X] should fail since the occur check detects a cycle.

Notice that structural representations permit the expression of higher order object level term and syntactic higher order unification. Consider for example the term [F,sy(X)], F is a meta-variable that should be unified to a term name standing for a unary object level functor.

3.4. Relating names and structural representations.

'Log provides the user with four infix operators <=p=>/2, <=c=>/2, <=t=>/2 and <=s=>/2 — called respectively the program, clause, term and symbol **correspondence operators** — to pass from one meta-representation to the other of the same syntactic entity.

Declaratively χ<=p=>υ (χ<=c=>υ, χ<=t=>υ, χ<=s=>υ) is true if χ is the program (respectively clause, term and symbol) name of an object and υ is the structural representation of the same object.

From an operational point of view, the goal

χ<=p=>υ (χ<=c=>υ, χ<=t=>υ, χ<=s=>υ)

succeeds if:

- χ is a program (respectively clause, term, symbol) name and it is possible to instantiate (the meta-variables occurring in) υ in such a way to obtain the structural representation corresponding to χ, or
- χ is a variable and υ is the ground structural representation of a program (respectively clause, term, symbol).

If both χ and υ contain meta-variables, the proof of the goal is postponed till either one of the above cases occur or no other goal in the current computation

can be further processed. In the second case, the destructuring operator is returned as part of the computed answer .

Here are some examples of goals involving <=*x*=>:

```
?- tr(f(g(a),b,C))<=t=>[sy(f),A,sy(b),sy(C)].      (1)
   A → [sy(g),sy(a)].

?- N<=t=>[sy(f),[sy(g),sy(a),sy(b),sy(C)].         (2)
   N → tr(f(g(a),b,C)).

?- N<=t=>[sy(f),A,sy(b),sy(C)], A=[sy(g),sy(a)].   (3)
   A → [sy(g),sy(a)],
   N → tr(f(g(a),b,C)).

?- N<=t=>[sy(f),A,B,sy(C)], A=[sy(g),sy(a)].       (4)
   A → [sy(g),sy(a)],
   N<=t=>[sy(f),[sy(g),sy(a)],B,sy(C)].
```

Delaying the solution of goals containing the <=*x*=> operator permits a declarative reading of 'Log programs. The order of literals in a clause or in a goal (e.g. in (3)) is immaterial; the commutativity of "," seen as the logical *and* connective is thus preserved. Moreover, even though it is not possible to completely solve the call of <=*x*=> (i.e., some variables in it are not instantiated yet) the computation does not fail. On the contrary, the <=*x*=> is returned as part of the computed answer; it will be considered as a **constraint** on the values that its uninstantiated variables can assume. For example, in (4) there are obviously valid instances of the (meta-)variables `B` and `N`, but not all are viable.

3.5. The semantic framework.

'Log possesses all of the semantic properties of a pure logic programming language as described, for instance, in [Lloy87]. Moreover, 'Log has been

proven in [Cerv92] to be an instance of the Constraint Logic Programming languages scheme (see [JaLa87a, JaLa87b, JaLM86a, JaLM86b, JaMi, LiSt]). Therefore, in this section, only the results which appear to be the most important to an implementation of 'Log will be presented. Greater details can be found in [Cerv91].

The main differences between 'Log and the standard case are due to the presence of the <=*x*=>/2 operators (that will be generically denotated as <=>, since all of them share the same theoretical properties). <=> is an interpreted operator. It establishes a relation between two syntactic expressions. When these expressions are not fully instantiated, the term containing <=> is returned to represent a **constraint** on values of its contained meta variables. The variable range in the computed substitution is restricted in accordance with the constraints on them.

In the CLP framework, 'Log is based on a structure whose domain is the (extended) Herbrand universe ***H*** and in which primitive relations are the equality (=) and the declarative reading of the <=> operators.

A term having the form χ<=>υ, where χ and υ are terms and **<=>** is its main functor is called an **R-term**. An R-term χ<=>υ is a **proper R-term** if there is a ground variable substitution θ such that χ^{θ} is a name and υ^{θ} is the corresponding structural representation, i.e. if χ^{θ}<=>υ^{θ} is true; otherwise it is **improper**. An R-term χ<=>υ having variables in both χ and υ is an **open R-term**. If exactly one between χ and υ has a variable in it, we have a **semi-ground R-term**. Finally, if neither χ nor υ contain variables, χ<=>υ is a **ground R-term**. R-terms are constraints Therefore, according to the standards of CLP, a 'Log clause can be rewritten as an object in the following format: H :- **C**, **B**. where H is the head, **C** is a conjunction of constraints and **B** is a conjunction of (user-defined) atoms.

The set $\Re$ of all proper ground R-terms, or **R-universe**, will play an important role in the following discussion. $\Re$ is an infinite subset of the Herbrand universe ***H***. Membership of an element to $\Re$ can be easily proved to be decidable [Cerv91]. ***H*** is defined as usual, except for being built out of a set of characters, the alphabet, instead of the set of predicate and function symbols that explicitly appear in a program.

The notion of **interpretation** of a 'Log program differs from the standard notion of Herbrand interpretation only for the request that the R-universe is always a subset of the set of the ground terms which are valid in the interpretation. The definitions of validity, model and logical consequence are given in the usual way. Then, it is possible to define a minimal model M_P for a given program P similarly to the standard case [Cerv91]. It is also possible to characterize a program P by means of a functional T_P defined as follows: for each interpretation I,

$$T_P(I) = \Re \cup \{A \in \boldsymbol{H} : A \text{ :- } A_1, \ldots, A_n \text{ is a ground instance of a clause of P and } A_1, \ldots, A_n \in I\}$$

where $\Re$ is the R-universe. T_P is monotonic and it can be proved to be continuous too. Finally, the minimal model of P can be proved to be equivalent to the least fixpoint of T_P [Cerv91].

Now that the declarative interpretation of a 'Log program has been discussed, its operational semantics can be described as well as the soundness and completeness results that relate the two semantics to each other.

A **reificator** is a finite (possibly empty) conjunction $R_1 \wedge \ldots \wedge R_n$ of proper open R-terms that is satisfiable, that is for which there exists a (ground) substitution θ such that $(R_1 \wedge \ldots \wedge R_n)^\theta$ is true in every interpretation (i.e., for each i=1..n, $R_i^\theta \in \Re$). Let P be a 'Log program and G = :-**C**,**B** be a goal. An answer to $P \cup \{G\}$ is a pair (θ,R), where θ is a substitution for G and R is a reificator such

that no variable occurring in R is in the domain of θ. A correct answer to P∪{G} is an answer (θ,R) such that

$$P \models (R \rightarrow (\mathbf{C},\mathbf{B})^{\theta})^{\forall}.$$

The following result [Cerv91] characterizes the relationship existing in 'Log between correct answers, validity and least model.

Theorem 1.

Let P be a 'Log program, G = :-**C**,**B** a goal and (θ,R) an answer for P∪{G} such that $\Re \models R$ and $(\mathbf{C},\mathbf{B})^{\theta}$ is ground. Then the following statements are all equivalent:

1. (θ,R) is a correct answer to P∪{G}
2. $(\mathbf{C},\mathbf{B})^{\theta}$ is a logical consequence of P
3. $(\mathbf{C},\mathbf{B})^{\theta}$ is true in M_P.

Let's now define a variant of the SLD-resolution method suited to 'Log program execution. Let G = :-A_1,...,A_m,...,A_n be a goal and C = B:-B_1,...,B_q a clause. The goal G' is (immediately) **derived** from G and C using the substitution θ if:

- θ is an mgu of the selected literal A_m and B (note that they could be both variables), and G' is the goal :-$(A_1,...,B_1,...,B_q,...,A_n)^{\theta}$, or
- A_m is the proper ground or semi-ground R-term χ<=>υ, θ is the mgu of υ and the structural representation corresponding to the name χ if χ is ground, or θ is the mgu of χ and the name corresponding to the structural representation υ if χ is not ground, and G' is the goal :-$(A_1,...,A_{m-1}, A_m,...,A_n)^{\theta}$. In this case, C will be assumed to be the empty clause.

Let P be a program and G a goal. A **derivation** d of P∪{G} is a (finite or infinite) sequence of triples $<(G_0,C_0,\theta_0),(G_1,C_1,\theta_1),...>$ such that G_0=G and for each i > 1, G_i is immediately derived from G_{i-1} using C_{i-1} and θ_{i-1}.

A **refutation** r of $P \cup \{G\}$ is a finite derivation $<(G_0,C_0,\theta_0),...,(G_n,C_n,\theta_n)>$ such that G_n is a reificator. The **computed answer** obtained by the refutation of $P \cup \{G\}$ is the pair (θ,G_n) where θ is the restriction of the composition $\theta_1...\theta_n$ to variables occurring in G.

This definition of refutation is more general than the usual one. Indeed, any Prolog refutation can be considered as a 'Log refutation which never applies a derivation step using R-terms and always ends up with the empty reificator.

The previous definitions have also been given in a CLP-like manner in [Cerv92]. The more traditional definitions have been reported here because they describe the operational behavior of an interpreter for the language in a more explicit way.

The soundness theorem for the 'Log resolution method can now be stated in exactly the same way as for the standard SLD-Resolution. A proof can be found in [Cerv91].

Theorem 2 (Soundness of the resolution method).
Let P be a program and G a goal. Every computed answer for $P \cup \{G\}$ is a correct answer for $P \cup \{G\}$.

To establish the completeness of 'Log's resolution method a further definition is required first. Let $R = R_1 \wedge ... \wedge R_n$ and $Q = Q_1 \wedge ... \wedge Q_m$ be two reificators. R is **more general** than Q (or R subsumes Q) if R imposes looser constraints than Q on the variables occurring in both Q and R, that is, more formally, if there exists a substitution θ such that for each i = 1,..,n, either $R_i^{\theta} \in \Re$ or $R_i^{\theta} = Q_j$ for some j = 1,..,m, and for each i = 1,..,n, if Q_j contains a variable appearing in R, then there is an i = 1,...,n such that $R_i^{\theta} = Q_j$.

Theorem 3 (Completeness of the resolution method)
Let P be a program and G a goal. If (θ,R) is a correct answer for $P \cup \{G\}$, then there exists a computed answer (σ,Q) for $P \cup \{G\}$ and a substitution γ such that $\theta = \sigma\gamma$ and Q is more general than R.

As in the standard case, it is possible to prove that the selection rule has no influence on the existence of a refutation for a given $P \cup \{G\}$. In this sense, it is possible to state a strong completeness theorem that allows to arbitrarily constraint the way the selected atom is chosen at each derivation step (in particular, to delay the proof of a goal containing <=>).

3.6. More 'Log.

In this chapter, only the core of 'Log has been described. 'Log is in effect much more. Even though nothing more that what has been presented so far will be needed in following discussion, it is worth giving an idea of the other features of 'Log.

The syntax adopted in this chapter is substantially different from what is described in [Cerv91, CeRo91a, CeRo91b, CeRo92]. In these papers, the basic notation for names is much more user oriented: for example, the term name of `f(g(a),b,C)` is `'f(g(a),b,C)'` rather than `tr(f(g(a),b,C))`. The new syntax has been introduced to facilitate the parsing process. In fact, the previous syntax was not even LR(1) [AhSU] while the adopted one is. It is quite easy to build a preprocessor to pass from one syntax to the other. In the previously cited papers, a user-friendly *synthetic notation* was defined for structural representations too. This was a rather natural way of expressing the second meta-representation of an object level entity, handling also the case of incomplete structural representations. For example, the synthetic form of the previous exam-

ple term is `"f(g(a),b,C)"` instead of `['f',['g','a'],'b','C']` (this form existed too, under the name of *explicit notation*). The meta-variables were prefixed by #'s. This syntax has been dropped too for simplicity. Once more, a simple preprocessor can be used to handle it.

Writing real meta-programs require the use of much more powerful tools than the correspondence operator. In particular, the equivalent of Prolog's `call/1`, `clause/2`, `assert/1` and `retract/1` [StSh] together with the `demo/2` predicate of [BoKo] are needed. Declarative versions of these predicates can be easily built in 'Log [Cerv91]. Unfortunately, relying on a 'Log implementation of these predicates can be very slow. It would be useful to implement directly these operators at a lower level, as predefined functions hardwired into the compiler or interpreter of 'Log. This has not been done since it was beyond the scope of this thesis.

4. Implementation of ’Log.

4.1. Introduction.

Even though ’Log is a simple variation of pure Prolog, the few features few added invalidate the well-established results upon which Prolog is founded. The same observation applies also as far as the implementation issue is concerned.

The ’Log compiler described here has been developed an Ethernet network of SUN3, DECstations 3100 and 5000. The operating system was ULTRIX V4.2A (Rev. 47). The whole system has been written in C. A WAM abstract machine for pure Prolog was initially constructed as a variant of the descriptions found in [AïtK91a] and [AïtK91b]. With that working system as a basis, the implementation of the features that properly belong to ’Log double size of the program.

Augmenting a pure Prolog system in order to accept and correctly execute ’Log requires introducing mechanisms to cope with the following novel aspects:

- finding an appropriate representation for the names in the system;
- finding an appropriate representations for the structural representations;
- devising a correct implementation of the <=*x*=> subgoals, that takes into account delaying these goals when necessary;

- enforcing the consistency on the pending <=*x*=> atoms;
- remembering the constraints still alive at the end of the execution.

Each of these tasks is easy in itself, but in order to achieve an acceptable degree of efficiency in the resulting 'Log system, each item in the above list must be chosen with extreme care.

The next section will describe the way the representation issue has been handled. Section 4.3 will focus on constraints and in particular to the way open R-terms are delayed and awaken when they become instantiated. The last section of this chapter corresponds to the user's manual of the implemented system.

4.2. Internal representation of the names.

Let's initially consider only the **representation issue**, i.e. the problem of accommodating names and structural representations on the heap of the WAM. Note first of all that once a representation schema has been chosen for names, the way structural representations are implemented is decided too. In fact, a structural representation is defined as a list of names; therefore, it is first of all a list and should be handled as such. So, the only point is to chose a good heap representation for names.

Names are subject essentially to two operations: unification and destructuring using the <=*x*=> operators. Therefore, it is wise to devise a representation for these entities that permit a fast execution of both of these operations. Unfortunately, efficient implementations of unification and destructuring place opposing demands on the heap representation of names.

In fact, in order to have a fast unification procedure, the representation of a name should be as compact as possible, possibly a reference to a symbol table (as for constants and functors). However, with such a representation, destructuring would be a burden, each time `<=x=>` should transform a name into the corresponding structural representation, this latter object must be constructed on the heap, even if just one element of this list is actually needed.

To allow fast destructuring, it is better for the two meta-representations of an object to be as similar as possible. In particular, a name should be represented as a list starting with a distinguished marker. Note that in this way, a name would be a special list of names, that in turn would be special lists of names, and so on until the level of character names is reached. This method has obvious drawbacks: first of all it makes use of a huge amount of memory, (i.e. heap cells) In fact, cells are needed for every character in the name to be represented plus extra cells for the multi-list structure. Also, name unification becomes a major source of overhead; in fact matching even simple names can require visiting of very big trees.

The solution adopted is a compromise between these two extreme proposals. The idea is to represent names as special lists, but only till the symbol name level. Symbol names are represented as atomic objects and normally occupy just one heap cell. Whenever the `<=s=>` operator is invoked, a corresponding special list of character names is built on the heap in the place where the implicit representation of the symbol name was held. Therefore, a symbol name is destructured only when this is really needed.

The implementation of this idea has required the introduction of seven new types of data cells (and therefore, seven new tags) and of eighteen new WAM instructions. These will now be described for each kind of name available in ’Log.

Character names require a single cell for their representation (also because these entities do not have a structural representations associated with them). The data cell tag for a character name is CHN while the value field of the cell is simply the ASCII code of the character itself (see figure 4.1). This new type of cell is very similar to the cell for the constants and so are the WAM instructions needed to handle this representation:

- *put_chn c, A_i* is supposed to be used when compiling a character name occurring as an argument of the body of a clause. It simply loads A_i with a CHN cell for c;

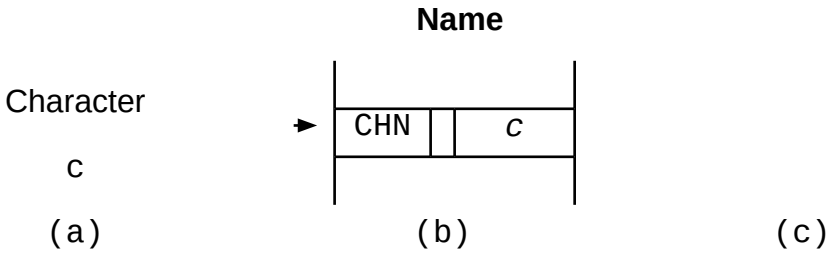

Figure 4.1 Heap representation of a character name

- *get_chn c, A_i* is used for character names occurring as arguments in the head of a clause. If A_i points to such an object, this character is checked for being c itself. If A_i points to an unbound reference cell, it is set to a CHN cell for c;
- *set_chn c* used when encountering an inner character name in the body of a clause, this instruction pushes a CHN cell for c on the top of the heap;
- *unify_chn c* in READ mode, it checks if the currently examined cell on the heap is a CHN cell for c; if it is instead an unbound reference cell, it binds it to such a cell for c. In WRITE mode, it behaves as set_chn.

The implementation choices previously stated make the handling of symbol names a non-trivial task. In fact, it requires the introduction of three new types of data cells and eight new instructions. This is shown in figure 4.2.

The name of a normal Prolog symbol can have two forms: an implicit form in which a single data cell is needed for its representation, and an explicit representation as the list of its constituting character names. Note that the same symbol name can appear in both forms during the execution of a program. In fact, the use of `<=c=>` allows one to dynamically pass from one form to the other. Note also that the unification algorithm must be modified so that the implicit and explicit form of the same symbol name do unify. In particular, whenever the explicit form is not completely specified (i.e. when it contains meta-variables), the unification procedure should be able to construct parts of it from the implicit representation. So, symbol names are a point where the unification and destructuring procedures interact very closely.

The way implicit symbol names are handled closely resembles to the way character names have been dealt with. The new tag SIN is introduced and the value field of the corresponding heap cell contains a direct representation of the symbol, in the same way as constants are represented inside constant cells.

The following new instructions are required to deal with implicit symbol name cells:

- *put_sin s,* A_i is supposed to be used when compiling a symbol name occurring as an argument of the body of a clause. It simply loads A_i with a SIN cell for s;

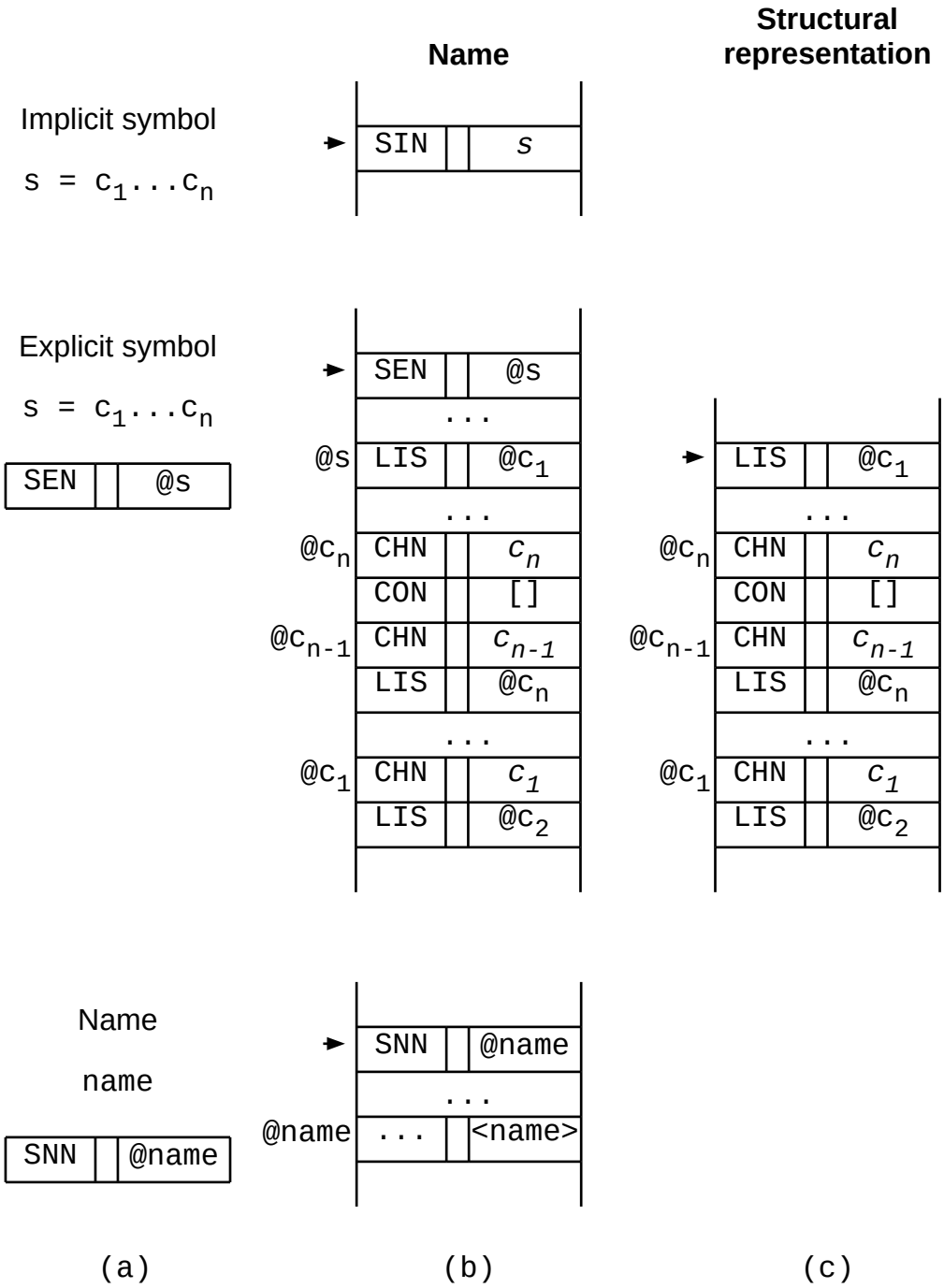

Figure 4.2 Heap meta-representations of a symbol

- *get_sin s, A_i* is used for symbol names occurring as arguments in the head of a clause. If A_i points to such an object, this symbol is checked for being s itself. If A_i points to an unbound reference cell, it is set to a SIN cell for s;
- *set_sin s*: used when encountering an inner symbol name in the body of a clause, this instruction pushes a SIN cell for s on the top of the heap;
- *unify_sin s* in READ mode, it checks if the currently examined cell on the heap is a SIN cell for s; if it is instead an unbound reference cell, it binds it to such a cell for s. In WRITE mode, it behaves as set_sin.

The explicit representation of a symbol is the list of its constituent character name cells, headed by a special SEN cell, where SEN is the new data tag introduced to handle this type of name. The value field of this kind of cells is a heap pointer to the head of the character name list. The way a symbol name is represented is shown in figure 4.2. Two new instructions are needed to handle SEN cells, they are conceptually similar to the two list cell instructions.. The compiler inserts these instructions in the object code of a 'Log program only when the symbol name appears on the left-hand side of a <=c=> subgoal.

- *put_sen X_i* It inserts in X_i a SEN cell pointing to the next available heap location.
- *get_sen X_i* It checks if the location pointed by X_i is a SEN cell and in this case, the next heap cell to visit is set to the cell pointed by it. If instead it is an unbound reference cell, it is bound to a new SEN cell pointing to the next available location on the heap and the mode is set to WRITE.

By definition, a name is a symbol, therefore the name of a name is a special kind of symbol name. In order to deal with this possibility, a third kind of

symbol name cell has been introduced, it is tagged as SNN and has a pointer to the first cell of the heap representation of the object name as its value field. The actual representation can be seen in figure 4.2. Two new instructions handle SNN cells and their behavior is similar to the SEN instructions:

- *put_snn* X_i It inserts in X_i a SNN cell pointing to the next available heap location.
- *get_snn* X_i It checks if the location pointed by X_i is a SNN cell and in this case, the next heap cell to visit is set to the cell pointed by it. If instead it is an unbound reference cell, it is bound to a new SNN cell pointing to the next available location on the heap and the mode is set to WRITE.

A term structure can be either the list of the structural representation of its components or just a symbol structure if the term it represents is a simple constant, a variable or a name. This is also the way term names are represented. A header TRN heap cell is used to indicate the starting point of a term name. As in the case of explicit symbol names, the value field of a TRN cell is a heap pointer to the first cell of the representation of the corresponding term structure. The way a term name is compiled is shown in figure 4.3. Two instruction are required to cope with TRN cells:

- *put_trn* X_i inserts in X_i a TRN cell pointing to the next available heap location.
- *get_trn* X_i checks if the location pointed by X_i is a TRN cell and in this case, the next heap cell to visit is set to the cell pointed by it. If instead it is an unbound reference cell, it is bound to a new TRN cell pointing to the next available location on the heap and the mode is set to WRITE.

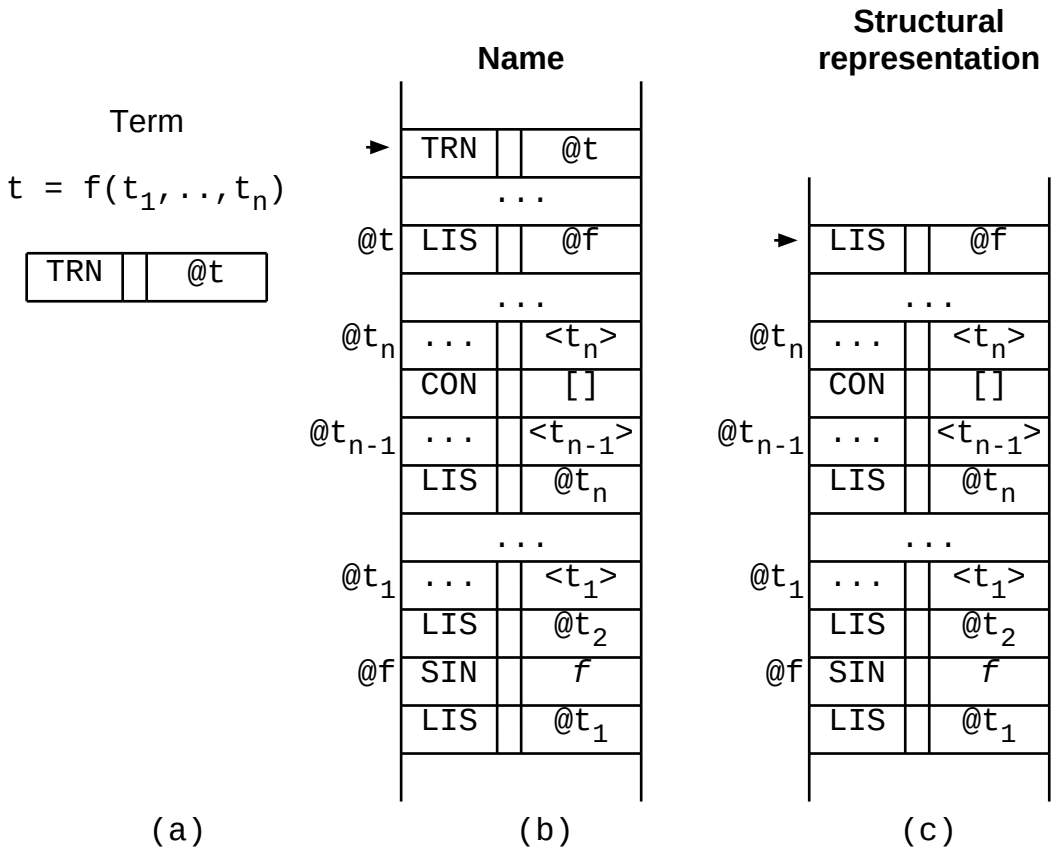

Figure 4.3 Heap meta-representations of a term.

Clause names are represented as the list of their constituent atoms. This list is leaded by a new data cell, tagged as CLN and having a pointer to the first cell of such a list. This is illustrated in figure 4.4. Similarly to term cells, handling CLN cells necessitates the introduction of two new instructions into the WAM instruction set:

- *put_cln* X_i inserts in X_i a CLN cell pointing to the next available heap location.
- *get_cln* X_i checks if the location pointed by X_i is a CLN cell and in this case, the next heap cell to visit is set to the cell pointed by it. If instead it is an

unbound reference cell, it is bound to a new CLN cell pointing to the next available location on the heap and the mode is set to WRITE.

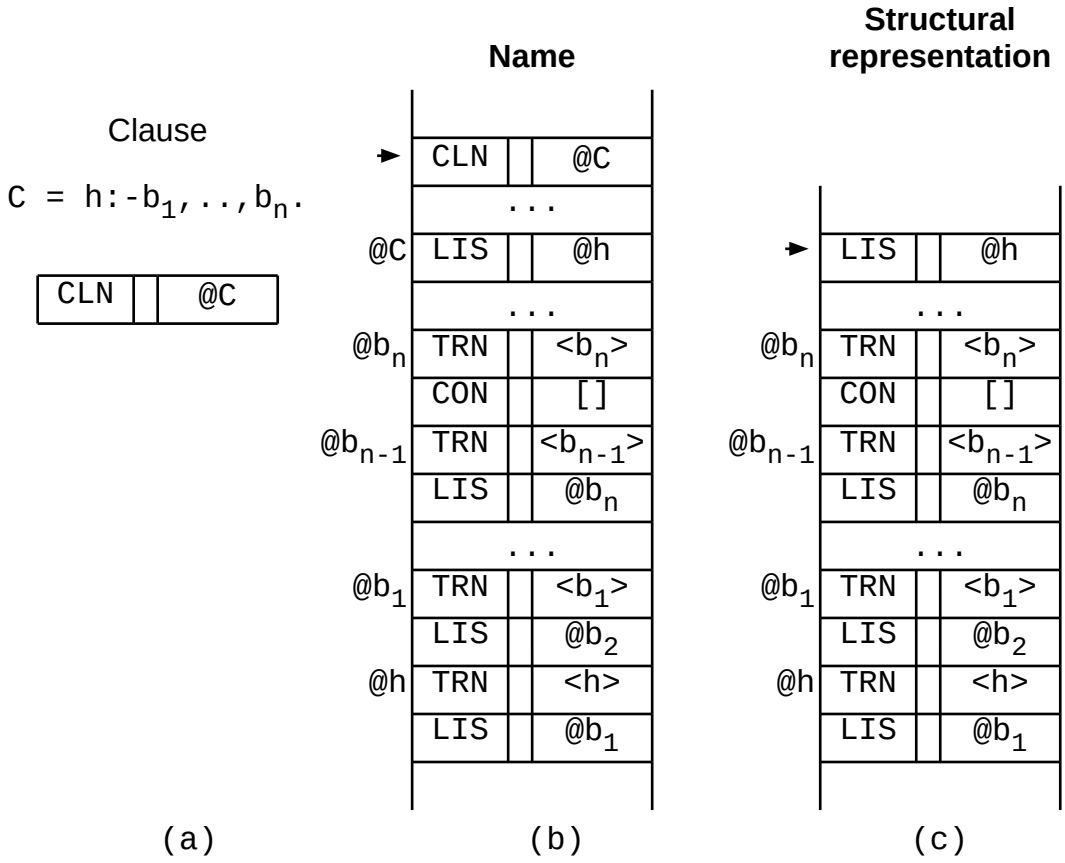

Figure 4.4 Heap meta-representations of a clause.

The way program names are implemented follows the lines set up for clauses, terms and explicit symbol names. A new data cell type, tagged PGN, is introduced to mark the clause list of the corresponding structural representation as a program name. Figure 4.5 illustrates this process. As in the previous cases, two new instructions are needed:

- *put_pgn* X_i inserts in X_i a PGN cell pointing to the next available heap location.

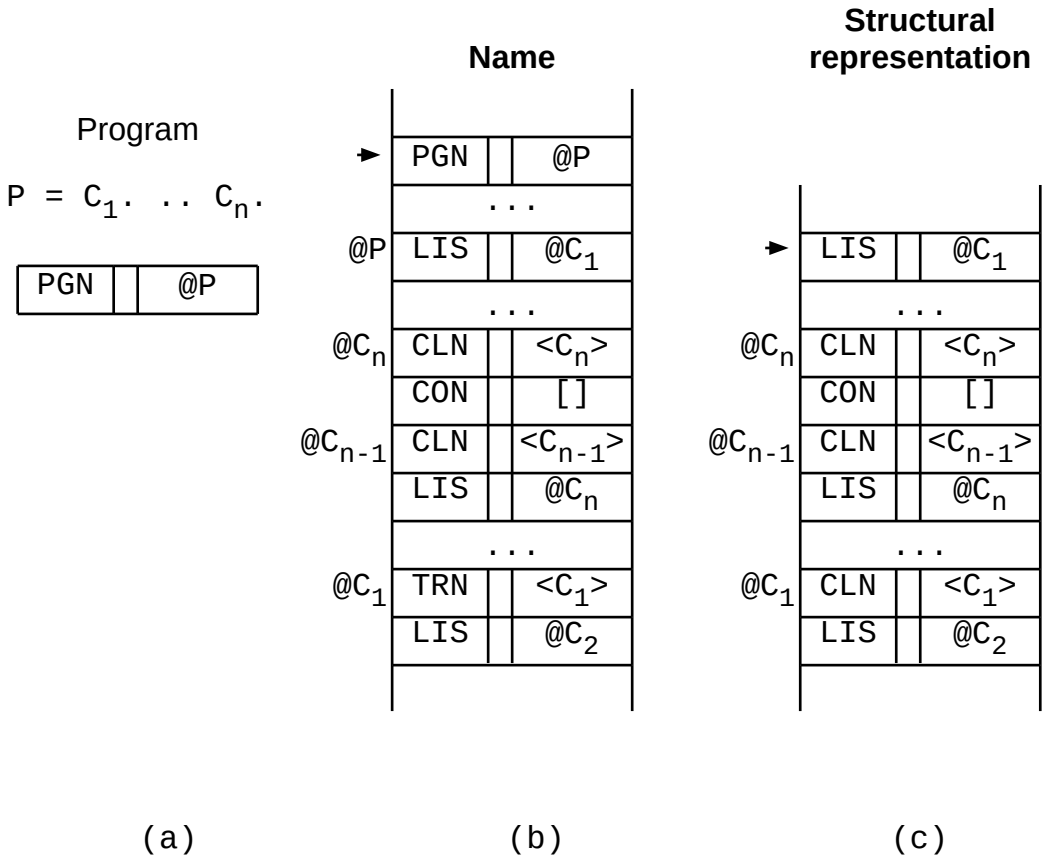

Figure 4.5 Heap meta-representations of a program.

- *get_pgn* X_i checks if the location pointed by X_i is a PGN cell and in this case, the next heap cell to visit is set to the cell pointed by it. If instead it is an unbound reference cell, it is bound to a new PGN cell pointing to the next available location on the heap and the mode is set to WRITE.

Figure 4.6 describes the way the term name `tr(foo(alpha,g(bb),X))` is compiled both when it occurs (a) in the head and (b) in the body of a clause and (c) the heap representation that it produces.

Notice how object level variables are treated exactly as the other symbols of the language.

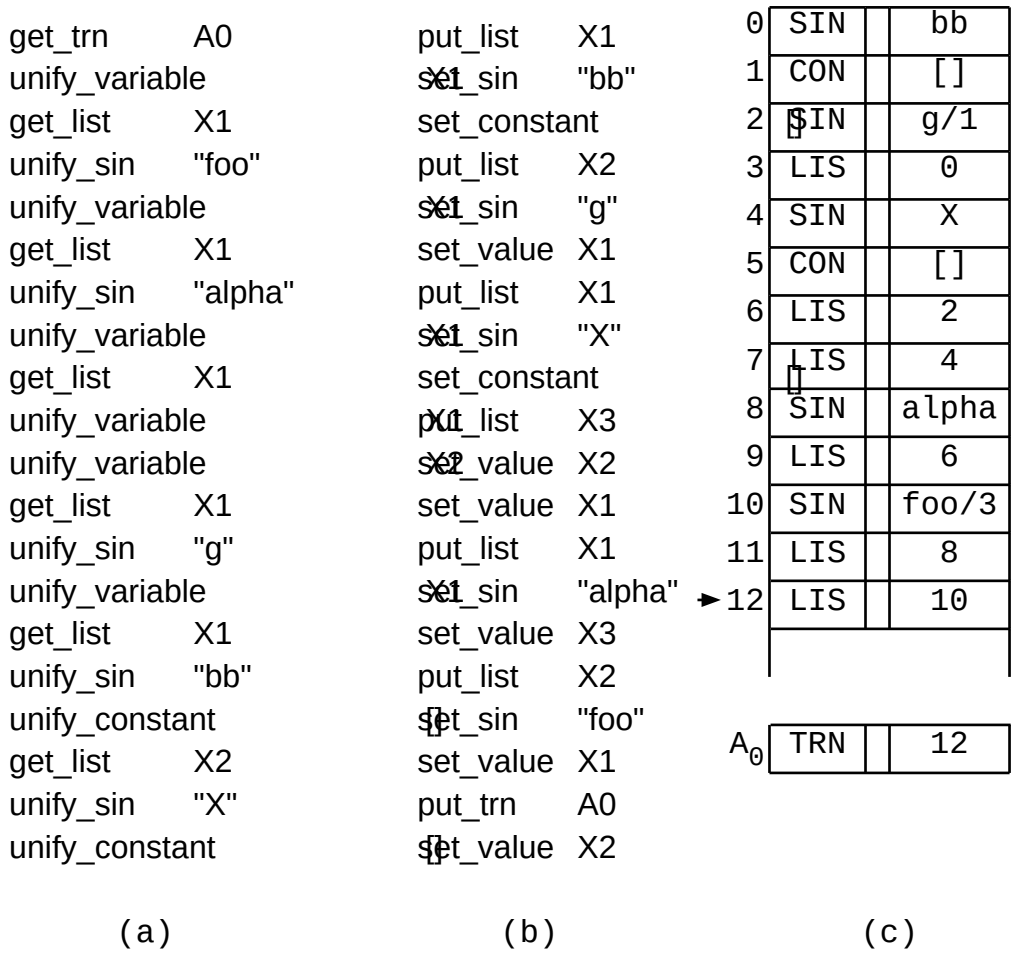

Figure 4.6 Compilation and heap representation of `tr(foo(alpha,g(bb),X))`.

4.3. Constraints.

Constraints and constraint languages have been around for many years. This topic has been studied intensively since the early 80's and has yielded to the definition of a constraint programming paradigm [Lele]. Constraints have

recently been introduced in logic programming as a useful generalization. Two main results have been obtained: the definition of the CHIP constraint logic programming language [DVS*, VanH] and the much more general definition of the CLP schema [JaLM86a, JaLM86b, JaMi, JaLa87a, JaLa87b]. In all these proposals, the constraints are mainly numerical, i.e. the user is allowed to program with integer or real equations that are interpreted as constraints for the variables appearing in them [Lele, Sanj]. 'Log's constraints are symbolic. However, some work has also been applied to these particular constraints [HaEl, Nade, DOPR].

Constraint handling has two aspects. First the *static*, or *syntactic issues* are concerned with the compilation of a constraint when it is encountered in the body of a clause. This aspect should be dealt with by the compiler. Second, since constraints cannot usually be solved immediately, there is the *dynamic issue*, handled by the executer, of monitoring the active constraints of the system. As unification instantiates the variables contained in the not completely solved constraints present in the system, they can become either inconsistent or solved. In the former case, a failure must occur. When a constraint becomes solved, it is not necessary to bother about it anymore.

The closeness of the physical representation of the name and the structural representation of a 'Log object has been exploited to obtain an elegant compiling schema format for constraints. In fact, since a name is in general no more than the corresponding structural representation prefixed by a name marker, the idea is to use the same heap space for the two members of a <=*x*=> operator.

When executing χ <=*x*=> υ, the heap representation of υ will initially be constructed and the register A_1 will be set to point to it. Then, a new name cell of the appropriate type will be pushed on the heap and set to point to the representation of υ too; A_0 keeps track of this cell. Figure 4.7 shows the execution process up to this point. χ can then be *read* from the content of the register A_0. In

this way, the code for χ will operate directly on the heap structure built for υ, eventually instantiating unbound variables contained in the (incomplete) structural representation υ.

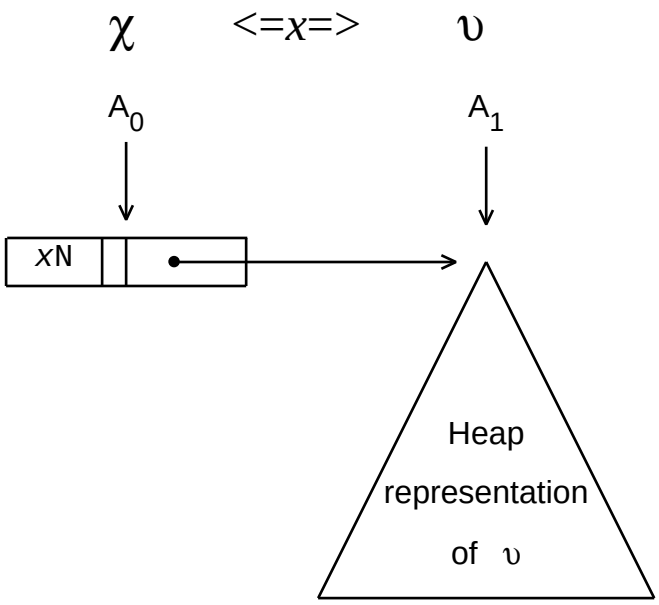

Figure 4.7 Heap configuration after solving χ <=x=> υ.

Once this process has been performed, A_0 can point either to a name, to an object containing variables that could become a name (if υ was an incomplete structural representation and χ a variable), or to an object that will never be able to become a name (if χ itself was incorrect). Therefore, the consistency of what is pointed by A_0 must be checked. This is the only novel aspect of the exectution, and to obtain it, four new instructions must be added to the WAM instruction set. They are:

- `pg_constraint` checks if register A_0 is pointing to a correct (eventually non completely specified) program name representation.
- `cl_constraint` checks if register A_0 is pointing to a correct (eventually non completely specified) clause name representation.

- *tr_constraint* checks if register A_0 is pointing to a correct (eventually non completely specified) term name representation.
- *sy_constraint* checks if register A_0 is pointing to a correct (eventually non completely specified) symbol name representation.

One more instruction has been added for improving the efficiency of the execution. It should be put at the beginning of the instruction sequence for a constraint. Its effect can be ignored.

- *watch_constr* sets an internal variable for an extra speed-up.

Figure 4.8(a) shows how a constraint of the form χ <=*x*=> υ is normally compiled. Some attention must be paid when χ is a previously seen variable. In fact, this variable could have already been instantiated. Therefore, its content should be copied in A_0 and a name cell must be retrieved from it rather than inserted. Once this has been done successfully, the value referenced by this cell should be unified with the content of A_1, i.e. the corresponding structural representation. This case is illustrates in Figure 4.8(b).

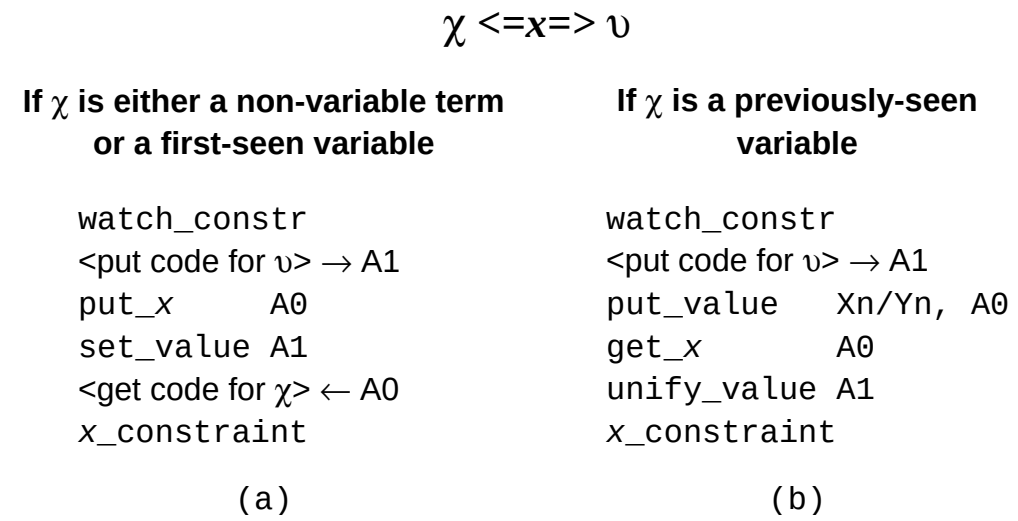

Figure 4.8 Compilation of χ <=*x*=> υ.

It is evident that the system must keep track, at run-time, of the constraints that have not yet been solved, called *active constraints*. Moreover, since any unification step can make some of them solved or inconsistent, the constraints present in the system should be periodically checked. When a constraint becomes inconsistent, backtracking must occur. When it gets solved, efficiency reasons suggest to remove it from the set of the active constraints. Unfortunately, this solution is too simplistic to work. In fact, successive failures may reactivate a previously solved constraint. Therefore, all constraints encountered during the execution must be retained, and inactive constraints will be marked as such. Ss for environments and choice points, the semantics of backtracking suggest that the most appropriate structure for recording constraints is a queue.

Therefore, a new queue has been added to the architecture of the WAM. It is called CTR and each of its elements describe a constraint χ <=*x*=> υ. A *constraint descriptor* has two fields: a tag telling if the constraint has been solved yet, and a pointer to the heap representation of χ. Each time a new constraint is encountered, the *x*_`constraint` instruction ending its compiled code pushes a descriptor on CTR. The tag field is eventually marked as solved.

It has been chosen to check the active constraints in the queue at the end of each rule execution. Therefore, `execute` and `proceed` have been modified to go through the CTR queue and check all the active constraints. During this process, a constraint can still be active and have been solved in which case its tag field is set to solved, or have become inconsistent in which case a failure must occur. An alternative solution was to monitor the unbound variables involved in a constant for their instantiation. This solution is better since it checks only the constraint which status could have changed. Unfortunately keeping track of which variables occur in which constraints is a non trivial task and does not

harmonize well with the WAM philosophy. Therefore, this solution has been discarded.

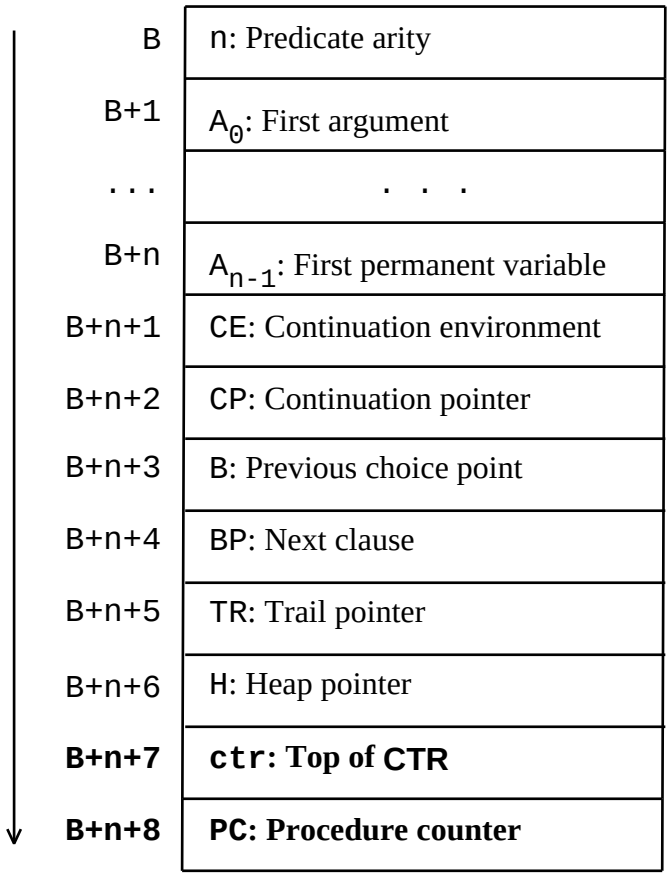

Figure 4.9 Choice point.

The most delicate point is how to modify the backtracking procedure so that when a failure occurs, all the operations done on the CTR queue are undone. First of all, when a new choice point is allocated on the stack, the current height of CTR must be recorded in it so that the successively added constraints can be discarded. A more serious question is how to remember which solved constraints to reset to active. This problem is solved with the introduction of a new internal register, called PC, or *procedure counter*. PC constantly contains the

current height of the resolution tree, i.e. the number of clause instances that are currently active. It is basically incremented by `call` decremented by `execute` and `proceed`. Now, whenever a choice point is allocated, the value of PC is recorded in it. The idea is to use the tag field of a constraint descriptor as a time-stamp. When a constraint is initially encountered, its tag field is set to zero, meaning unsolved. When the constraint is finally solved, its tag field is set to the current value of PC. Now, if a failure occurs, PC is reset to the value contained in the last available choice point, that is lower than the value it had when the failure occurred. Among other things, CTR is reset to the height it had at the creation time of the choice point and all the remaining portion of this queue is examined: all the solved constraints with a tag field containing a value higher than the current content of PC are reset to unsolved, i.e. to zero.

Figure 4.9 show the content of a 'Log choice point including he new fields introduced for constraint handling.

4.4. Summary.

Let's now summarize the WAM concepts resulting from this chapter. Figure 4.10 shows the new instructions added to the standard WAM,. It completes the schema of figure 2.22 where the instructions for the classical WAM where summarized. Figure 4.11 illustrates the internal data structures utilized by the extended WAM.

Head		Body	
	Program names		
get_pgn	X_i/Y_i	put_pgn	X_i/Y_i
	Clause names		
get_cln	X_i/Y_i	put_cln	X_i/Y_i
	Term names		
get_trn	X_i/Y_i	put_trn	X_i/Y_i
	Symbol names		
get_sen	X_i/Y_i	put_sen	X_i/Y_i
get_sin	s, A_i	put_sin	s, A_i
unify_sin	s	set_sin	s
get_snn	X_i/Y_i	put_snn	X_i/Y_i
	Symbol names		
get_chn	c, A_i	put_chn	c, A_i
unify_chn	c	set_chn	c
	Constraint handling		
		wach_constr	
		pg_constraint	
		cl_constraint	
		tr_constraint	
		sy_constraint	

Figure 4.10 The new WAM instruction set.

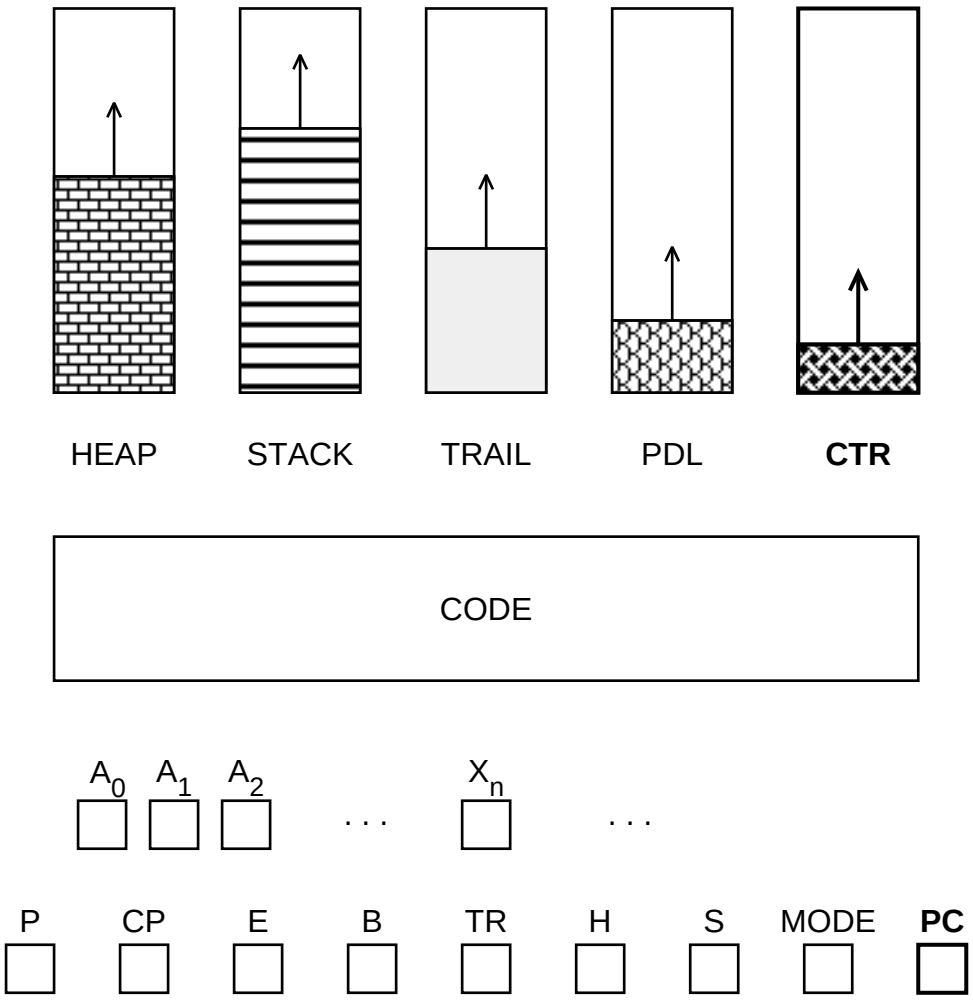

Figure 4.11 The WAM working areas and registers.

4.5. User manual.

The WAM system that has been implemented looks like a classical command line executer for Prolog. It is invoked by typing

wam [*file_name*]

at the system prompt, where *file_name* is an optional file name. The system enters its command mode with the following prompt:

wam>

If a file name has been specified, it has automatically been compiled and the resulting code has been loaded into the code area.

At this point the following commands are available:

:- <query>.	Runs a query.
compile(*file_name*).	Compiles and load *file_name*.
edit(*file_name*).	Edits *file_name*; then compiles and loads it.
load(*file_name*).	Loads *file_name*.
save(*file_name*).	Saves the current program to *file_name*.
listwam(*file_name*).	Produces an object listing of the current program.
debug.	Enters DEBUG mode.
nodebug.	Exits DEBUG mode.
shell.	Invokes an operating system shell.
! <*OS command*>.	Runs an operating system command.
help.	Describes the commands.
quit.	Exits the system.

When entering the DEBUG mode, the

debug>

prompt appears and the set of available commands is extended to allow the user to trace the execution of the current program at the WAM code level and to inspect the content of all the WAM data structures. These commands are described when by the on-line help.

5. Conclusions.

A technique for compiling the meta-programming logic programming language ’Log has been presented. In particular, the extensions to the standard Warren Abstract Machine (WAM) architecture necessary to support the execution of these programs have been described.

Meta-programming is a programming technique in which a problem is expressed partly at an object level and partly at a meta-level that operates syntactically on the object level representation. ’Log is a logic programming language upgraded with meta-programming capabilities. It introduces two meta-representations for each syntactic entity and a set of operators to manipulate them at the meta-level. This language still possesses the declarative semantics of pure Prolog.

Upgrading the WAM architecture to support ’Log has required to cope with some novel implementation challenges. First, an efficient internal representation for both meta-representation of ’Log must be provided. Second the execution of a ’Log program generates constraints that must be handled dynamically by the run-time architecture of the WAM. None of these problems is present in Prolog. A ’Log compiler has been implemented along with a WAM based run-time support architecture. The efficiency of the resulting system is comparable to that of a WAM for conventional Prolog.

Having a ’Log machine opens new perspectives for ’Log. In fact this language may be thought as an assembly language in the environment of meta-

programming in logic programming. Some extensions have already been proposed. They will can now easily been implemented in ’Log in order to test their usability. The author has in mind also some upgrades to ’Log itself. In particular, a more usable syntax should be introduced. Another interesting topic is studying the possibility to introduce reflection mechanisms into ’Log. A more ambitious goal is making a clear analysis of the relations between meta-programming and Gödel’s theorems, which lay in some sense at the origin of meta-programming.

References.

[AhSU] Aho A.V., Sethi R., Ullman J.D.: "*Compilers, Principles, Techniques and tools*", Addison-Wesley, 1986.

[AiCS] Aiello L., Cecchi C., Sartini D.: "Representation and Use of Metaknowledge" in *Proceedings. of the IEEE*, vol 74, n. 10, Oct. 1986, pp 1304-1321.

[AiLe] Aiello L., Levi G.: "The Use of Metaknowledge in IA Systems", in "*Meta-Level Architectures and Reflection*" (Maes P., Nardi D. Eds.), North-Holland, 1988, pp 243-254.

[AïtK91a] Aït-Kaci H.: "Warren's Abstract Machine: a tutorial reconstruction", *8th International Conference in Logic Programming* (ICPL'91), Paris, 1991.

[AïtK91b] Aït-Kaci H.: "*Warren's Abstract Machine*", MIT Press, 1991.

[Bach] Bacha H.: "Meta-Level Programming: a Compiled Approach", in "*Proceedings of the Fourth International Conference on Logic Programming*" (Lassez J.-L., ed), pp 394-410, the MIT Press, 1987.

[Bowe] Bowen K.A.: "Meta-Level Programming and Knowledge Representation" in *New Generation Computing* 3 (1985), pp 359-383.

[BBCT] Bowen K.A., Buettner K.A., Cicekli I., Turk A.K.: "The Design and Implementation of a High-speed Incremental Portable Prolog Compiler", in

"*Proceedings of the Third International Conference on Logic Programming*" (Shapiro E., ed), pp 650-656, the MIT Press, 1986.

[BoKo] Bowen K.A., Kowalski R.A.: "Amalgamating Language and Metalanguage in Logic Programming", in "*Logic Programming*" (Clark K.L., Tärnlund S.Å. Eds.), Academic Press, 1982, pp 153-172.

[BoWe] Bowen K.A., Weinberg T.: "A Meta-Level Extension to Prolog" in IEEE Symposium on Logic Programming, Boston 1985, pp 669-675.

[Brat] Bratko I.: "*Prolog - Programming for Artificial Intelligence*", Addison-Wesley, 1986.

[Buet] Buettner K.A.: "Fast Decompilation of Compiled Prolog Clauses", in "*Proceedings of the Third International Conference on Logic Programming*" (Shapiro E., ed), pp 663-670, the MIT Press, 1986.

[Carl] Carlsson M.: "Freeze, Indexing, and other Implementation Issues in the WAM", in "*Proceedings of the Fourth International Conference on Logic Programming*" (Lassez J.-L., ed), pp 40-58, the MIT Press, 1987.

[Cerv91] Cervesato I.: "*Una proposta per l'introduzione di capacita' di meta-rappresentazione in un linguaggio di programmazione logica*" (in Italian), Tesi di Laurea in Science dell'Informazione, Universita' degli Studi di Udine, March 1991.

[Cerv92] Cervesato I.: Private communication, electronic mail to G.F. Rossi, May 1992.

[CeRo91a] Cervesato I., Rossi G.F.: "Meta-programmazione logica in 'Log" (in Italian), *VI Convegno Nazionale sulla Programmazione Logica* (GULP'91), Pisa, Italy, June 1991.

[CeRo91b] Cervesato I., Rossi G.F.: "Logic Meta-programming in ’Log", *internal report n. RR/14/91*, dipartimento di Informatica e Matematica, Universita’ degli Studi di Udine, 1991.

[CeRos92] Cervesato I., Rossi G.F.: "Logic Meta-Programming facilities in ’Log", accepted for publication at the *3rd Workshop on Meta-Programming in Logic (Meta-92)*, Uppsala (Sweden), June 1992.

[Cost] Costantini S.: "Semantics of a Metalogic Programming Language" in "*Proceedings of the 2nd Workshop on Meta-Programming in Logic*" (Bruynooghe M., Ed.), Leuven (Belgium), April 1990, pp 3-18.

[CoDL] Costantini S., Dell’Acqua P., Lanzarone G.A.: "Estensioni di Ordine Superiore a Prolog sono Necessarie" (in Italian), in "*Atti del Quinto Convegno Nazionale sulla Programmazione Logica* (GULP 90)" (Bossi A. Ed.), 1990, pp 167-183.

[CoLa] Costantini S., Lanzarone G.A.: "A Metalogic Programming Language.", in "*Logic Programming, Proceedings of the Sixth International Conference*" (G. Levi, M. Martelli, Eds), MIT Press, 1989, pp 218-233.

[DellA] Dell’Acqua P.: Private communications, June 1992.

[DeMC] Demoen B., Marien A., Callebaut A.: "Indexing Prolog Clauses", in "*Proceedings of the North American Conference on Logic Programming*" (Lusk E., Overbeek R. eds.), MIT Press, vol. 2, pp 1001-1012, 1989.

[DVS*] Dincbas M., Van Hentenryck P., Simonis H., Aggoun A., Graf T., Berthier F.: "The Constraint Logic Programming Language CHIP", in "*Proceedings of the International Conference on Fifth Generation Computer Systems*", ICOT, pp 693-702, 1988.

[DOPR] Dovier A., Omodeo G.O., Pontelli E., Rossi G.F.: "Embedding Finite Sets in a Logic Programming Language", to be published in the "*Proceedings of Workshop on Extensions of Logic Programming*" (WELP92)", Springer-Verlag, Bologna, 1992.

[Ende] Enderton H.B.: "*A Mathematical Introduction to Logic*", Academic Press, Inc., 1972.

[Ferr] Ferrarelli A.: "*Implementazione di un interprete per un Prolog esteso con capacita' di meta-programmazione logica*", (in Italian), Tesi di Laurea in Science dell'Informazione, Universita' degli Studi di Udine, June 1991

[GaLa] Gallaire H., Lasserre C.: "Metalevel Control for Logic Programs", in "*Logic Programming*" (Clark K.L., Tärnlund S.Å. Eds.), Academic Press, 1982, pp 173-185.

[HaEl] Haralick R.M., Elliott G.L.: "Increasing Tree Search Efficiency for Constraint Satisfaction Problems", in *Artificial Intelligence*, Vol 14, pp 263-313, 1980.

[HiLl88] Hill P.M., Lloyd J.W.: "Analysis of Meta-Programs", *Technical Report CS-88-08*, Department of Computer Science, University of Bristol, 1988.

[HiLl91] Hill P.M., Lloyd J.W.: "The Gödel Report (Preliminary Version)", *Technical Report n. TR-91-02*, University of Bristol, Department of Computer Science, March 1991.

[JaLa87a] Jaffar J., Lassez J.L.: "Constraint Logic Programming" in "*Acts of the 14th POPL*", Münich, West-Germany, January 1987, pp 111-119.

[JaLa87b] Jaffar J., Lassez J.L.: "From Unification to Constraints" in "*Proceedings of the Fifth Conference on Logic Programming*" (Kowalski R.A., Bowen K.A. eds.), The MIT Press, 1987.

[JaLM86a] Jaffar J., Lassez J.-L., Maher M.J.: "Logic Programming Language Scheme", in "*Logic Programming: Functions, Relations and Equations*" (De Groot D., Lindstrom G. eds), pp 441-468, 1986.

[JaLM86b] Jaffar J., Lassez J.-L., Maher M.J.: "Some Issues and Trends in the Semantics of Logic Programming", in "*Proceedings of the Third International Conference on Logic Programming*" (Shapiro E., ed), pp 223-241, the MIT Press, 1986.

[JaMi] Jaffar J., Michaylov S.: "Methodology and Implementation of a CLP System", in "*Proceedings of the Fourth International Conference on Logic Programming*" (Lassez J.-L., ed), pp 196-218, the MIT Press, 1987.

[Klee] Kleene S.C.: "*Introduction to Metamathematics*", North-Holland, 1952.

[KluSz] Kluzniak F., Szpakowicz S.: "*Prolog for Programmers*", Volume 24 of A.P.I.C. studies in Data Processing, Academic Press, 1985.

[Kurs] Kursawe P.: "How to Invent a Prolog Machine", in *New Generation Computing*, 5, pp 97-114, 1987.

[Lele] Leler Wm.: "*Constraint Programming Languages: their Specification and Generation*", Addison-Wesley, 1988.

[LiSt] Lim P., Stuckey P.J.: "Meta-Programming as Constraint Programming", in "*Proceedings of the 1990 North American conference on Logic Programming*" (Debray S.K., Hermenegildo M., Eds.), 1990, pp 416-430.

[LiOK] Lindholm T., O'Keefe R.A.: "Efficient Implementation of a Defensible Semantics for Dynamic Prolog Code", in "*Proceedings of the Fourth International Conference on Logic Programming*" (Lassez J.-L., ed), pp 21-39, the MIT Press, 1987.

[Lloy87] Lloyd J.W.: "*Foundations of Logic Programming*", Springer-Verlag, II ed., 1987.

[Lloy88] Lloyd J.W.: "Directions for Meta-Programming", in "*Proceedings of the International Conference on Fifth Generation Computer Systems*", Tokyo, Nov. 1988, pp 609-617.

[MaNa] Maes P., Nardi D.: "*Meta-Level Architectures and Reflection*", North-Holland, 1988.

[MaRo] Martelli A., Rossi G.F.: "Enhancing Prolog to support Prolog Programming Environment", in *ESOP '88* (Ganzinger H. Ed.), L.N.C.S. 300, Springler Verlag, 1988, pp 317-327.

[MiNa86] Miller D., Nadathur G.: "Higher-order Logic Programming", in "*Proceedings of the Third International Conference on Logic Programming*" (Shapiro E., ed), the MIT Press, 1986, pp 448-462.

[MiNa87] Miller D., Nadathur G.: "A Logic Programming Approach to Manipulating Formulas and Programs", in "*Proceedings of the 1987 IEEE Symposium on Logic Programming*", San Francisco, 1987, pp 381-388.

[Nade] Nadel B.A.: "Constraint Satisfaction Algorithms", in *Computational Intelligence* 5, National Research Council of Canada, Ottawa, 1989, pp 188-224.

[Ross89] Rossi G.F.: "Meta-Programming Facilities in an Extended Prolog", in "*Artificial Intelligence and Information-Control Systems of Robots*" (Plander I. Ed.), North-Holland, 1989.

[Ross90a] Rossi G.: "Programs as Data in an Extended Prolog", *Internal Report n. RR-07-90*, University of Udine, Department of Computer Science, 1990.

[Ross90b] Rossi G.: "Note sull'Uso e la Definizione di un Linguaggio per la Programmazione Logica Strutturata" (in Italian), in "Atti del Quinto Convegno Nazionale sulla Programmazione Logica (GULP 90)" (Bossi A. Ed.), 1990, pp 185-199.

[Ross91] Rossi G.F.: Private communications. October 1991.

[Sanj] Sanjai N.: "*LOG(F): an Optimal Combination of Logic Programming, Rewriting and Lazy Evaluation*", The Rand Corporation, April 1988.

[Ster] Sterling L.: "The Paradigm of Meta-Programming" in "*Tutorial notes of the Proceedings of the 2nd Workshop on Meta-Programming in Logic Programming*" (Bruynooghe M., Ed.), Leuven (Belgium), April 1990, pp 1-26.

[StBe] Sterling L., Beer R.D.: "Incremental Flavor-Mixin of Meta-Interpreter for Expert System Construction", in "Proceedings of the 1986 IEEE Symposium on Logic Programming", Salt Lake City, 1986, pp 20-27.

[StSh] Sterling L.S., Shapiro E.: "*The Art of Prolog*", MIT Press, 1986.

[Suga] Sugano H.: "Meta and Reflective Computation on Logic Programming and Its Semantics", in "Proceedings of the 2nd Workshop on Meta-

Programming in Logic Programming" (Bruynooghe M., Ed.), Leuven (Belgium), April 1990, pp 19-34.

[VanH] Van Hentenryck P.: "*Constraint Satisfaction in Logic Programming*", the MIT Press, 1989.

[Warr] Warren D.H.D.: "An Abstract Prolog Instruction Set", *Technical Note 306*, SRI International, Menlo Park, CA, 1983.

[Weyh] Weyhrauch R.W.: "Prolegomena to a Theory of Mechanized Formal Reasoning" in Artificial Intelligence 13 (1980), pp 133-170.

A WAM IMPLEMENTATION FOR THE META-LOGIC PROGRAMMING LANGUAGE 'LOG

A Thesis

Presented to

the Faculty of the Department of Computer Science

University of Houston

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

By

Iliano Cervesato

August, 1992

A WAM IMPLEMENTATION FOR THE
META-LOGIC PROGRAMMING LANGUAGE ’LOG

_____

Iliano Cervesato

APPROVED:

_____

Dr. L. P. Slothouber, Chairman

_____

Dr. J. C. Huang, Co-Chairman

_____

Dr. K. Kaiser

_____

Dean, College of Natural Sciences and Mathematics

Acknowledgments

This thesis is just the the top of an iceberg that has grown for past two years. Not cold it is, but full of intense human relationships. Therefore, I would like to remember all the people who have crossed my life during this period, contributing to make me the person I am. I would like to thank them for having been there. In particular, the following persons have got a particular place in my heart: all the members of the "italian group" for having been my family for two years and in particular Barnabás Takács, Paola Vesentini, Enrico Pontelli and Stefano Gallucci; Ingrid and Claudia Peñarrieta, Jana Fritsch, Vincent Bricout, Patricia Saavedra, Elisa Guardia.

A special thank you goes to Enrico Pontelli for his contribution to writing the first version of the Prolog WAM-based compiler and for the useful discussions we had. I must also mention Prof. Gianfranco Rossi for having introduced me to the topic of meta-programming and for his valuable contribution to the definition of ’Log.

I would like to express my gratitude to my advisor, Dr. L. P. Slothouber for the freedom he left me during the development of this thesis and for the patience he proved to have. I am also grateful to Dr. J. C. Huang and Dr. K. Kaiser for serving as members of my thesis committee.

The attendance of the Master of Sciences in Computer Science at the University of Houston has been possible thanks to my sponsor, the European Social Fund, and the School of Master in Computer Science held by IAL in Pordenone, Italy. I want in particular to thank the staff there with who I shared so many working nights: Mauro Bresin and Clementina Carnera.

Last but not least, I am particularly debtful to all the people at the Computer Science department of the Universita’ di Torino, Italy, for allowing me to complete this Master before entering their Ph.D. program as a full-time student. In particular, Prof. Alberto Martelli, Maria Luisa Sapino and Laura Giordano will bear my gratitude forever for the help they gave me, probably beyond their own imagination.

Thank y’all,

Iliano Cervesato, Houston, TX, July 17th 1992.

A WAM IMPLEMENTATION FOR THE
META-LOGIC PROGRAMMING LANGUAGE 'LOG

Abstract of a Thesis
Presented to
the Faculty of the Department of Computer Science
University of Houston

In Partial Fulfillment
of the Requirements for the Degree
Master of Science

By
Iliano Cervesato
August, 1992

Abstract

A technique for compiling the meta-programming logic programming language ’Log is presented and the extensions to the standard Warren Abstract Machine (WAM) architecture necessary to support the execution of these programs are described. Meta-programming is a programming technique in which a problem is expressed partly at an object level and partly at a meta-level that operates syntactically on the object level representation. ’Log is a logic programming language upgraded with meta-programming capabilities. It introduces two meta-representations for each syntactic entity and a set of operators to manipulate them at the meta-level. This language still possesses the declarative semantics of pure Prolog. Upgrading the WAM architecture to support ’Log has required to cope with some novel implementation challenges. First, an efficient internal representation for both meta-representation of ’Log must be provided. Second the execution of a ’Log program generates constraints that must be handled dynamically by the run-time architecture of the WAM. None of these problems is present in Prolog. A ’Log compiler has been implemented along with a WAM based run-time support architecture. The efficiency of the resulting system is comparable to that of a WAM for conventional Prolog.

Contents

1. Introduction. 1

2. The Warren Abstract Machine. 7

2.1. Introduction. 7

2.2. Basic unification. 8

2.2.1. WAM data representation. 11

2.2.2. WAM instructions 12

2.2.3. Compilation of queries. 15

2.2.4. Compilation of programs 16

2.3. Facts. 17

2.4. Flat resolution. 20

2.5. Pure Prolog. 25

2.6. Optimizations. 29

2.6.1. Constants. 30

2.6.2. Lists 32

2.6.3. Registers 34

2.6.4. Miscellaneous stack optimizations. 34

2.7. Summary. 37

3. Logic meta-programming in 'Log. 39

3.1. Introduction. 39

3.2. Names. 42

3.3. Structural representations. 44

3.4. Relating names and structural representations. 46

3.5. The semantic framework. 47

3.6. More 'Log. 52

4. Implementation of 'Log. 54

4.1. Introduction. 54

4.2. Internal representation of the names. 55

4.3. Constraints. 65

4.4. Summary. 71

4.5. User manual. 73

5. Conclusions. 75

References. 77

List of Figures

Figure 2.1 Architecture of the WAM. 7

Figure 2.2 Steps of the unification algorithm. 10

Figure 2.3 Data cells and term representation. 13

Figure 2.4 Query code for `:-p(Z,h(Z,W),f(W)).` 13

Figure 2.5 Program code for `p(f(X),h(Y,f(a)),Y).` 14

Figure 2.6 Code pattern for a query and a program fact. 18

Figure 2.7 Code for `:-p(Z,h(Z,W),f(W)).` 19

Figure 2.8 New program code for `p(f(X),h(Y,f(a)),Y).` 19

Figure 2.9 Control-correct code pattern for facts and rules. 22

Figure 2.10 Environment frame. 24

Figure 2.11 Code pattern for facts and rules. 24

Figure 2.12 WAM code for `p(X,Z):-r(X,Y),q(Y,Z).` 25

Figure 2.13 Choice point. 27

Figure 2.14 Code pattern for procedure definitions. 28

Figure 2.15 WAM code for append/3. 29

Figure 2.16 Compilation and heap representation of `f(b(g(a)).` 31

Figure 2.17 Enhanced compilation and heap representation of `f(b(g(a))`. 32
Figure 2.18 Compilation and heap representation of `[a,b,c]`. 33
Figure 2.19 Enhanced compilation and heap representation of `[a,b,c]`. 33
Figure 2.20 Code for `append/3` before and after improving register allocation. 35
Figure 2.21 Definitive WAM code for `append/3`. 36
Figure 2.22 The WAM instruction set. 37
Figure 2.23 The WAM working areas and registers. 38
Figure 4.1 Heap representation of a character name 57
Figure 4.2 Heap meta-representations of a symbol 59
Figure 4.3 Heap meta-representations of a term. 62
Figure 4.4 Heap meta-representations of a clause. 63
Figure 4.5 Heap meta-representations of a program. 64
Figure 4.6 Compilation and heap representation of `tr(foo(alpha,g(bb),X))`. 65
Figure 4.7 Heap configuration after solving χ <=*x*=> υ. 67
Figure 4.8 Compilation of χ <=*x*=> υ. 68
Figure 4.9 Choice point. 70
Figure 4.10 The new WAM instruction set. 72
Figure 4.11 The WAM working areas and registers. 73