AN EXTENDED WARREN ABSTRACT MACHINE FOR THE EXECUTION OF STRUCTURED LOGIC PROGRAMS

EVELINA LAMMA, PAOLA MELLO, AND ANTONIO NATALI

▷ Extending logic programming towards structuring concepts such as modules, blocks, taxonomy of logic theories and viewpoints leads, from the implementation point of view, to the development of more complex, specialized execution models to achieve acceptable efficiency.

In this work we address the effective implementation of a general framework, subsuming standard Prolog, where different languages for structuring logic programs can be efficiently supported and integrated and thus become useful constructs to build real applications.

The implementation is based on an extension of the abstract machine (WAM) developed by D. H. D. Warren, obtained by adding a new stack, new registers, and some new instructions to the WAM.

In the paper we focus on the design of the extended WAM. Moreover, some performance results are discussed. ◁

1. INTRODUCTION

A crucial research topic of logic programming is how to introduce structuring concepts. Some proposals extend logic programming with concepts very similar to those of procedural languages such as modules and blocks [14, 15, 27]. Other proposals, integrating logic and object-oriented programming, introduce inheritance between separate logic theories [4, 11, 13, 16, 22, 36]. The dynamic composition of logic theories, instead, has been proposed to build viewpoints and perform hypothetical reasoning [4, 12, 21].

Address correspondence to E. Lamma, P. Mello, or A. Natali, Dipartimento di Elettronica, Informatica e Sistemistica, Università di Bologna, Viale Risorgimento, 2 40136 Bologna, Italy. E-mail: {evelina, paola, natali}@deis33.cineca.it.

This work has been partially supported by the Progetto Finalizzato Sistemi Informatici e Calcolo Parallel of CNR under grant n. 92.01606.PF69.

Accepted January 1991.

THE JOURNAL OF LOGIC PROGRAMMING

655 Avenue of the Americas, New York, NY 10010 0743-1066/92/$5.00

It should therefore be of great interest in defining a *general framework* on the basis of which various proposals for structuring logic programs can be well described, integrated and efficiently implemented.

In more detail, by "general framework" we mean a set of simple concepts added to logic programs and mapped in a few operational mechanisms that, suitably combined, can be used to build different structuring constructs at language level.

A first step in this direction can be found in [3] and [24], where a general framework that subsumes and integrates some of the best-known proposals for structuring logic programs is designed and its declarative and operational semantics is formalized.

In this paper we focus more deeply on implementation issues. We show that this general framework can be efficiently implemented by easily extending well-known techniques used for standard logic programming.

The general framework, in fact, is implemented on the basis of an extended, or *structured*, Warren Abstract Machine [33] (hereinafter called S-WAM) supporting different structured logic programming languages. The S-WAM maintains full compatibility with Prolog. In the S-WAM the memory organization has been extended with respect to the standard WAM in order to support separate data bases and their static or dynamic combination. A new stack has been added along with two new instructions to expand or contract it. Moreover, the structure of both the choice point and the environment of the WAM have been expanded to consistently handle new registers. The real, efficient implementation of the S-WAM has been done in a compilation-based environment, based on a special-purpose VLSI microcoded architecture, [8,9] in the style of [28]. Block-, module-, inheritance-based logic programs, together with systems for hypothetical reasoning, can be implemented and integrated on this efficient architecture.

The paper mainly focuses on the S-WAM design and is organized as follows. In Section 2, the general framework supporting blocks, modules, object-oriented concepts and hypothetical reasoning is briefly introduced. In Section 3 some examples of structured logic programs are discussed. In Section 4, the operational semantics is given. In Section 5, the S-WAM design is discussed in depth and some optimizations and extensions introduced. In Section 6, the compilation-based environment is described and some performance results given. In Section 7 related works are discussed.

2. A GENERAL FRAMEWORK FOR STRUCTURING LOGIC PROGRAMS

In this section, we briefly present a general framework for structuring logic programs, suitable for implementing block-, module-, inheritance-based systems and hypothetical reasoning representation. A deeper discussion, together with a declarative characterization can be found in [24] and [3]. Here only a sketch is given in order to clearly understand the S-WAM design and the real implementation.

The basic idea originates from Contextual Logic Programming [26]. A structured logic program can be conceived as a collection of independent modules, called *units*. A unit consists of a set of clauses and is denoted by a unique atomic name. Units can be (possibly dynamically) connected into hierarchies (called *contexts*) which, in turn, provide the set of definitions for query evaluations. In this way, rather than evaluate a goal g by simply using a fixed and statically determined set

of clauses, as happens in pure Prolog, g is evaluated by using a variable set of clauses determined by the *current context of proof*.

As a matter of fact, contexts are represented as ordered lists of unit names and denote the union of the sets of clauses of the composing units. In particular, if the current context is $[U_N, U_{N-1}, \ldots, U_1]$ the ordered set of clauses considered is:

$\langle U_N$ clauses$\rangle$
$\langle U_{N-1}$ clauses$\rangle$
...
$\langle U_1$ clauses$\rangle$

The built-in predicate (*context-extension operator*) $U_N \gg G$, where U_N is the name of a unit and G a goal, forces the proof of G in a new context $C1$ obtained by conceptually "stacking" the U_N clauses on top of the previous context C. The underlying system guarantees automatic discarding of the U_N clauses at the end of the demonstration of G, both in the case of success and failure.

Example 2.1. Let us consider the following program:

unit(list1):
member(X,[Y|_]) :- eq1 ≫ equal(X,Y).
member(X,[_|Z]) :- member(X,Z).
unit(eq1):
equal(X,X).

The top goal: *list1 ≫ member(a, [a, b, c])* has the following top-down derivation [26]:

[ ] ⊢ list1 ≫ member(a, [a, b, c])
[list1] ⊢ member(a, [a, b, c])
[list1] ⊢ eq1 ≫ equal(a, a)
[eq1, list1] ⊢ equal(a, a)
success

A more generic definition of *member/2* can be obtained by omitting the context extension operator in its body:

unit(list2):
member(X,[Y|_]) :- equal(X,Y).
member(X,[_|Z]) :- member(X,Z).
permutation([ ],[ ]).
permutation(L,[X|P]) :- del(X,L,L1),permutation(L1,P).

The top goal: *eq1 ≫ list2 ≫ member(a, [a, b, c])* has the following top-down derivation:

[ ] ⊢ eq1 ≫ list2 ≫ member(a, [a, b, c])
[eq1] ⊢ list2 ≫ member(a, [a, b, c])
[list2, eq1] ⊢ member(a, [a, b, c])
[list2, eq1] ⊢ equal(a, a)
success

If one desires to use *member/2* with a different definition for *equal/2*, it is sufficient to call *member/2* with a different context, e.g., by using the top goal:

eq2 ≫ *list2* ≫ *member(a, [a, b, c])*,

where

```
unit(eq2):
equal(*,_).
equal(?,_).
equal(X,X).
```

Different policies for composing units into contexts can be adopted and, accordingly, different classes of structuring mechanisms can be identified, as discussed in [24] and [3]. In Figure 1, some proposals for structuring logic programs are classified in terms of the policies of unit compositions introduced in the following.

2.1. *Extension and Overriding for Predicate Definitions*

In logic programming nondeterminism is present since each *p/n* procedure may correspond to different clauses. In this setting two forms of nondeterminism are provided. A procedure, in fact, may have multiple definitions not only in the same unit (*intraunit nondeterminism*), but also in different units of the same context (*interunit nondeterminism*).

Two different policies can be adopted with reference to predicate definitions in the context:

(1) The most recent predicate definition *overrides* the previous ones for the same predicate. Only intraunit nondeterminism may be present.
(2) The most recent predicate definition *extends* the previous ones for the same predicate. Both intra- and interunit nondeterminism may be present. In the general framework, the default policy is predicate overriding. To obtain the predicate extension policy for predicate *p/n*, the following declaration must be inserted: **$extends(*p/n*)**. If this declaration is present in a unit *U*, not only the definition of *p/n* in *U*, but also those in units before *U* in the current context will be taken into account.

	Dynamically Configured	**Statically Configured**
Evolving Binding	Multi-Prolog [7] ***E*** N-Prolog [12] ***E*** Clausal Int. Logic [21] ***E*** Modules [25] ***E*** [3] ***E/O***	SPOOL [11]* ***E/O*** Class Temp. [22]* ***E/O*** [3] ***E/O***
Conservative Binding	Contexts [26] ***O*** Prolog/KR [29] ***O***	Meta-Prolog [2] ***E*** SPOOL [11] ***E/O*** Class Templ. [22]* ***E/O*** Blocks and Modules [15] ***E*** Prolog/KR [29] ***O***

*when using the label self

FIGURE 1. Classifying different proposals in the general framework (E stands for predicate extension, while O for predicate overriding).

Moreover, to support information hiding, a predicate definition p/n is exported from a unit U (i.e., visible outside it) only if the declaration, **$visible(*p*/*n*)**, is present in U. In the following, for the sake of simplicity, all predicates are assumed to be *visible*.

The use of the $extends declaration avoids replication of the code in some cases.

Example 2.2. All the occurrences of unit *eq*2 in Example 2.1 could be replaced by the context extension eq1 ≫ eq2_II, where eq2_II is:

```
unit(eq2_II):
$extends(equal/2).
equal(*,_).
equal(?,_).
```

2.2. *Conservative and Evolving Policies for Binding Predicate Calls*

The concept of context can be mapped in the concept of *binding environment*, a common notion in traditional programming languages to bind a name to a value for some period during execution of a program.

In order to evaluate a predicate call in a context we have to find, in that context, the appropriate set of definitions, i.e., the binding for predicate calls.

Two different policies of binding—referred to as *conservative* and *evolving* [3] can be adopted.

Let us suppose that $C = [u_N, \ldots, u_i, \ldots u_1]$ is the current context. If an *evolving policy* is adopted, the predicate definition for a call p occurring in unit u_i is given by the clauses of the whole context C. We will refer to C as the *global context (GC)*. If a *conservative policy* is adopted, the predicate definition for a call p occurring in unit u_i is given by the clauses of the subcontext $[u_i, \ldots u_1]$. We will refer to this subcontext as the *partial context (PC)*. Structured logic programming systems adopting a conservative policy are more static, efficient (see Section 5) and safe.

The general framework considered here supports both policies and adopts the conservative one as default. Predicate calls following the evolving policy are prefixed by the symbol #. In order to support both policies, two different contexts (i.e., the *global* and the *partial*) have to be maintained by the run-time support of the language (see Section 5.1). In this setting a top-down derivation (see the operational semantics, Section 4) is given in terms of sequences of formulae of the kind, $GCPC \vdash F$, where GC is the current global context, PC the current partial one and F a goal formula. In unit *list2* of Example 2.1, the predicate call for equal/2 follows a conservative policy.

Example 2.3. To better understand the difference between conservative and evolving policies let us consider the following unit:

```
unit(eq3):
equal(X,X).
equal([X|A],[Y|B]) :- permutation([X|A],[Y|B]).
```

The top goal: eq3 ≫ list2 ≫ member([a, b], [[b, a], c]) has the following derivation where the first list represents the global context GC, and the second represents the partial context PC:

[ ][ ] ⊢ eq3 ≫ list2 ≫ member([a, b], [[b, a], c])
[eq3][eq3] ⊢ list2 ≫ member([a, b], [[b, a], c])
[list2, eq3][list2,eq3] ⊢ member([a, b], [[b, a], c])
[list2, eq3][list2,eq3] ⊢ equal([a, b], [b, a])
[list2, eq3][eq3] ⊢ equal([a, b], [b, a])
[list2, eq3][eq3] ⊢ permutation([a, b], [b, a])
failure

The goal *permutation/2* in *eq3* is bound with respect to the partial context [eq3] and then fails. Changing the order of *eq3* and *list2* in the context does not solve the problem (*equal/2* will fail). The best solution, to avoid the explicit naming of the unit *list2* inside *eq3*, is to consider an evolving policy for *permutation/2* subgoal, i.e., considering the new code:

unit(eq4):

```
equal(X,X).
equal([X|A],[Y|B]) :- #permutation([X|A],[Y|B]).
```

The top goal: eq4 ≫ list2 ≫ member([a, b], [[b, a], c]) has the following top-down derivation:

[ ][ ] ⊢ eq4 ≫ list2 ≫ member([a, b], [[b, a], c])
[eq4][eq4] ⊢ list2 ≫ member([a, b], [[b, a], c])
[list2, eq4][list2,eq4] ⊢ member([a, b], [[b, a], c])
[list2, eq4][list2,eq4] ⊢ equal([a, b], [b, a])
[list2, eq4][eq4] ⊢ equal([a, b], [b, a])
[list2, eq4][eq4] ⊢ #permutation([a, b], [b, a])
[list2, eq4][list2, eq4] ⊢ del(b, [a, b], L1),permutation(L1, [a])
... success

In this case, the goal *permutation2* is solved with reference to the global context [list2, eq4] and the right definition is found in *list2*.

2.3. Building the Context

A context can be built by using the *context extension operator* (firstly introduced in [26]). If $C = [u_N, \ldots, u_1]$ is the current context, evaluation of the extension formula $u \gg G$ causes the proof of G to be performed in the new context $[u, u_N, \ldots u_1]$ obtained by adding unit u on top of the previous context C.

Since two contexts are taken into account in our framework, two different extension operators (≫ / ⋙, hereinafter referred to as *cactus/linear*) are provided. The goal $u \gg G$ extends the current partial context with unit u and then makes GC equal to PC. Conversely, goal $u \ggg g$ extends the current global context with unit u and then makes PC equal to GC (see the operational semantics in Section 4).

2.4. *Statically and Dynamically Configured Units*

In order to support a more static, efficient and safe view of structured logic programs, we now introduce two kinds of units, *dynamically* and *statically* configured, that produce different behaviours when extending the context.

When a *statically configured* (static for shortcut) unit is defined, a *fixed* context (hereinafter called *closure*) is statically associated with it. Whenever a *statically configured* unit u is asked for the proof of a goal, g, (e.g., by invoking goal $u \gg g$ or $u \ggg g$), the proof of g takes place in the closure of u, whatever the current context. A *dynamic configuration* on the other hand, provides a more flexible framework. In this case, no context is statically associated with the unit. Whenever a *dynamically configured* (dynamic for shortcut) unit u is asked for the proof of a goal g, the proof of g takes place in the current, dynamic context of the computation. Of course, static units are more efficient than dynamic ones, since a greater number of predicate calls can be completely solved in them at compile-time (see Section 5.3).

Example 2.4. Let us consider the top goal: eq2 ≫ list2 ≫ member(*,[a,b,c]), where *eq2* and *list2* of Example 2.1 are assumed to be dynamic. The following derivation will take place:

[ ][ ] ⊢ eq2 ≫ list2 ≫ member(, [a, b, c])*
[eq2][eq2] ⊢ list2 ≫ member(, [a, b, c])*
[list2, eq2][list2, eq2] ⊢ member(, [a, b, c])*
[list2, eq2][list2, eq2] ⊢ equal(, a)*
[list2, eq2] ⊢ equal(, a)*
success

Let us now suppose that unit *list2* must always use the definition of *equal/2* present in the unit *eq1* of Example 2.1. Then unit *list2_II* can be defined as a statically configured unit having the same source code as *list2*, and also as a "fixed" context [eq1]. For the top goal: eq2 ≫ list2_II ≫ member(*,[a,b,c]) the following derivation will take place:

[ ][ ] ⊢ eq2 ≫ list2_II ≫ member(, [a, b, c])*
[eq2][eq2] ⊢ list2_II ≫ member(, [a, b, c])*
[list2_II, eq1][list2_II, eq1] ⊢ member(, [a, b, c])*
[list2_II, eq1][list2_II, eq1] ⊢ equal(, a)*
[list2_II, eq1][eq1] ⊢ equal(, a)*
failure (backtracking)

The same effect is obtained by directly calling *eq1 ≫ equal(X, Y)* in *list1*, but the solution with the static unit *list2_II* is more efficient, as will be clear in Section 5.3.

In some cases it may be useful not only to extend the current context but also to *switch* to a new, different context. This effect can be directly obtained, without the introduction of a new operator, by using context extension and statically configured units.

It is sufficient to assume the existence of a primitive, statically configured unit, called *top*, with no predicate definition and having the empty context as closure.

3. EXAMPLES OF STRUCTURING POLICIES

Most of the proposals for structuring logic programs found in the literature can be included in the classification above. A detailed discussion about this topic is out of the scope of this paper and can be found in [3]. In the sequel, we briefly classify some systems with respect to the different kind of relationship among units (i.e., *static/dynamic*) and the different binding policies (i.e., *conservative/evolving*) and present some examples (see Figure 1). In particular, we discuss more deeply the case of inheritance-based systems and the handling of viewpoints when hypothetical reasoning is concerned.

3.1. Blocks and Modules

In block-based systems (see for example [15]), static scope rules determine predicate visibility on the basis of the nesting of blocks in the program. To prove an atomic goal occurring in a clause of a given block, only those clauses defined in that block or in enclosing blocks can be used. Such a behavior corresponds to a statically configured system with a conservative policy, where each block is encapsulated into a separate unit having the enclosing block as its *closure*.

In the case of closed modules (see [14]), proving a goal in a module M, means taking into account only those predicate definitions which are local to M. Closed modules can be still be classified as static conservative systems, where each unit always has an empty associated context and therefore uses only its local definitions. Examples of embedding blocks and modules in our language can be found in [3].

3.2. Inheritance and Object-Based Systems

Inheritance- and object-based systems (see for example [13, 22]) can be classified as statically configured evolving systems. We can interpret contexts as the explicit representation of a branch in an inheritance tree (for the sake of simplicity we will not consider multiple inheritance). The first unit in the context is the tip node, while the last one is the top of the hierarchy.

Inheritance-based systems are intrinsically evolving since, as stated in [35], "a self-reference in a type or class is bound to the object on whose behalf an operation (demonstration) is being executed, rather than on the textual module (unit) in which the self-reference occurs."

Example 3.1. Let us consider the *Class Template Language* described in [22]. When we say that a *bird* is a special case of *animal* we are stating that whatever holds for animals also holds for birds: the theory *bird* inherits from the theory *animal.*

In the *Class Template Language* we express this kind of relationship between classes by means of class rules:

bird $\Leftarrow$ animal; tweety $\Leftarrow$ bird
horse $\Leftarrow$ animal; person $\Leftarrow$ animal

where:

```
animal:   mode(walk).
          mode(run) :- self:no_of_legs(2).
          mode(gallop) :- self:no_of_legs(4).
bird:     mode(fly).
          no_of_legs(2).
          covering(feather).
horse:    no_of_legs(4).
human:    no_of_legs(2).
tweety:   no_of_wings(2).
```

The call *self:g* causes the proof of g to be performed in the tip class of the current hierarchy, no matter what the current class is. The use of *self* allows therefore to model the expected behaviour of inheritance. This behaviour is intrinsically evolving; in our framework this program can be translated into the following (static) evolving one, where *unit(u, static(list))* means that the unit u is associated with the transitive closure of the context specified by *list*.

```
unit(animal,static[top]):
mode(walk).
mode(run) :- #no_of_legs(2).
mode(gallop) :- #no_of_legs(4).
unit(bird,static([animal])):
mode(fly).
no_of_legs(2).
covering(feather).
unit(horse,static([animal])):
no_of_legs(4).
unit(human,static([animal])):
no_of_legs(2).
unit(tweety,static([bird])):
no_of_wings(2).
```

Now the taxonomy is embedded in the *static* declaration, e.g., the context associated with *tweety* will be *[tweety, bird, animal, top]* and the *self* behaviour is automatically expressed by the evolving policy (i.e., by the '#' operator).

Other examples, together with a discussion on how implementing multiple inheritance in this setting can be found in [23].

3.3. Viewpoints

Dynamic configurations, together with the *linear* extension operator and the *evolving* binding policy, provide a natural support for viewpoints as a simple form of hypothetical reasoning. In fact, extending the current context with a new unit

can be understood as adding a new hypothesis to the current line of reasoning. Then if a unit u embodies some hypotheses, evaluating the goal $u \ggg G$ means evaluating G after assuming the hypotheses in u. From this point of view, the *linear* extension operator is very similar to the *assume* predicate introduced in [34] and to the *hypothetical implication* introduced in [12], [21] and [25]. Systems handling viewpoints can be classified as dynamically configured evolving systems, where hypotheses or viewpoints are collected in units.

These systems are intrinsically evolving since the newly added knowledge always updates the old knowledge. Similarly, they have to be dynamically configured, in order for the new hypotheses or viewpoints to be added to the dynamic state of the computation.

Example 3.2. The following dynamically configured evolving program provides a structured representation of the well-known example of the block world.

```
unit(u0):
on(a,b).
on(b,c).
on(c,table).
next(X,Y) :- on(X,Y).
next(X,Y) :- on(Y,X).
colour(b,black).
next_w(B) :- next(B,X),colour(X,white).
unit(v1):
colour(a,white).
unit(v2):
colour(c,white).
unit(main):
next_white(B) :- u0 ⋙ v1 ⋙ next_w(B),
                 u0 ⋙ v2 ⋙ next_w(B).
```

The unit *u0* represents a given *state* of the world; units *v1* and *v2*, in turn, represent two possible viewpoints expressing the additional knowledge:

```
color(a,white) ∨ color(c,white).
```

In order to prove that there exists a block next to a white block, we use the definition for *next_w* in unit *main* which takes into account both the viewpoints *v1* and *v2*. Accordingly, the goal *u0* ⋙ *next_w(B)* succeeds with substitution $\{B/b\}$.

Some examples combining viewpoints and inheritance are also discussed in [4].

4. OPERATIONAL SEMANTICS

In this section we sketch the inference rules used by the extended Prolog machine supporting structured logic programs. The design of the S-WAM described in Section 5, is heavily inspired by these rules. For the sake of simplicity, $extends/$visible declarations are not taken into account. The Prolog-like syntax

of the overall general framework, given in the BNF notation, is the following, where terminal symbols are in bold, and *A* and *Constant* represent, respectively, atomic formulae and constants:

⟨*program*⟩ ::= **program[**⟨unit-sequence⟩**]**
⟨unit⟩ ::= **unit(**⟨*unit-name*⟩[,⟨*unit-type*⟩]**)**:⟨unit-code⟩
⟨unit-name⟩ ::= *Constant*
⟨*unit-type*⟩ ::= **static(**[⟨unit-sequence⟩]**)**|**static(**[ ]**)**|**dynamic**
⟨unit-sequence⟩ ::= ⟨*unit-name*⟩|⟨*unit-name*⟩,⟨*unit-sequence*⟩
⟨*unit-code*⟩ ::= ⟨s-clause⟩.|⟨s-clause⟩.⟨unit-code⟩
⟨s-clause⟩ ::= A :– ⟨*goal-formula*⟩
⟨*goal-formula*⟩ ::= ⟨single-goal-formula⟩|⟨single-goal-formula⟩,⟨goal-formula⟩
⟨single-goal-formula⟩ ::= ⟨*atomic-goal-formula*⟩|⟨*extension-formula*⟩
⟨*atomic-goal-formula*⟩ ::= **true**|A|#A
⟨*extension-formula*⟩ ::= ⟨unit-name⟩ ⋙ ⟨single − goal − formula⟩|
⟨unit-name⟩ ≫ ⟨single-goal-formula⟩

Let us now introduce the operational semantics of the overall general framework. In the sequel, we denote:

A, A′, atomic goal formulae;
g, a single-goal formula, i.e., an atomic or extension goal;
G, a conjunction of single-goal formulae;
ϵ, k, l, substitutions (ϵ is the empty substitution);
moreover, the composition of substitutions is denoted by juxtaposition;
{G}k is the application of the substitution k to the formula G;
mgu(A, A′) is the most general unifier of the atomic formulae A and A′;
unit clauses have the conventional body "true," which always holds.

Given a program *P* (viz. a set of units), we denote:

Units(P) = {u|u is a unit name of P};
|u| = {c|c is a clause in u};
C(P) = {ctx|ctx is a list of unit names in Units(P)}.

Given a program *P*, we denote by:

closure: Units(P) → *C(P)* a function that, given a unit U, returns the corresponding context associated with U (if U is static). In detail, *closure* can be defined in terms of an auxiliary function called *seq-closure* as follows:

closure (U) = [U|seq-closure(C)]

if U is static and C is the context associated with U.

$$\text{seq-closure(C)} = \begin{cases} [\ ] \text{ if C is empty.} \\ [\text{head(C)}|\text{seq-closure}(\text{tail(C)})], \text{ if head(C) is dynamic.} \\ \text{closure}(\text{head(C)}), \text{ if head(C) is static.} \end{cases}$$

Definition 4.1. (Top-down derivation). Let P be a program and G a goal formula. A top-down derivation of G in P can be traced in terms of (possibly infinite) sequences of steps of the kind $GC_i PC_i \vdash_{k_i} G_i$, where GC_i is the global context

and PC_i the partial one, k_i a substitution and G_i a goal formula, starting from both empty global and partial context. Each step is obtained by applying suitable inference rules. A top-down derivation is successful if, at some step n, the null formula is derived. □

We now give the set of inference rules describing the operational behavior of the general framework.

Definition 4.2 (Inference rules for the general framework).

TRUE:

$$(1)\ \frac{}{GC\,PC \vdash_{\epsilon} \text{true}}$$

CONJUNCTION:

$$(2)\ \frac{GC\,PC \vdash_{k} g;\ GC\,PC \vdash_{l} \{G\}k}{GC\,PC \vdash_{kl} (g, G)}$$

ATOMIC GOAL I:

$$(3)\ \frac{A' :\!-\ G \text{ is a definition for A in } |u|;\ k = mgu(A, A');\ GC[u|PT] \vdash_{l} \{G\}k}{GC[u|PT] \vdash_{kl} A}$$

ATOMIC GOAL II:

$$(4)\ \frac{\text{no definition exists for A in } |u|;\ GC\,PT \vdash_{k} A}{GC[u|PT] \vdash_{k} A}$$

ATOMIC GOAL (EVOLVING):

$$(5)\ \frac{GC\,GC \vdash_{kl} A}{GC\,PC \vdash_{kl} \#A}$$

CACTUS EXTENSION WITH DYNAMIC UNITS:

$$(6)\ \frac{u \in \text{Units(P) and is dynamic};\ [u|PC][u|PC] \mid -_{k} g}{GC\,PC \vdash_{k} u \gg g}$$

LINEAR EXTENSION WITH DYNAMIC UNITS:

$$(7)\ \frac{u \in \text{Units(P) and is dynamic};\ [u|GC][u|GC] \vdash_{k} g}{GC\,PC \vdash_{k} u \ggg g}$$

CACTUS/LINEAR EXTENSION WITH STATIC UNITS:

$$(8)\ \frac{u \in \text{Units(P)};\ u \text{ is static and closure } (u) = C;\ CC \vdash_{k} g}{GC\,PC \vdash_{k} u \gg g \text{ (or } u \ggg g)}$$

5. THE S-WAM

In this section, we present the abstract machine (designed as an extension to the Warren Abstract Machine [33] and referred to as Structured Warren Abstract Machine or S-WAM in the following) supporting units and their static or dynamic combination into contexts. In particular, the S-WAM supports and integrates static

and dynamic units, evolving and conservative predicate binding policies, and predicate extension or overriding.

A new dynamic data area (called *instance environment stack*) has been added to WAM, to represent the binding environment for eager and lazy predicate calls. New registers refer to the instance environment stack, and new instructions are also added to the WAM instruction set in order to expand and diminish this data area. Moreover, the structure of both the choice point and the environment of the WAM have been expanded to consistently handle new registers and some optimizations have also been considered to limit the overhead in the case of execution of standard Prolog programs.

The current WAM techniques used to generate the intermediate code have been straightforwardly extended in order to cope with units and contexts. The compiler classifies each predicate call occurring in a unit U as:

(1) *local*, if a local predicate definition for it exists in U;
(2) *eager*, if no local predicate definition for it exists in U and a conservative policy is adopted to search for its definition in the current (partial) context.
(3) *lazy*, if it is prefixed by the # operator. In this case an evolving policy is adopted to search for its definition in the current (global) context (see Section 2.2).

The difference between these goals and, therefore, their names can be interpreted in terms of *binding time*. Bindings for local goals can be solved at compile-time. Bindings for eager goals of a unit U can be solved:

- when the current context is extended with unit U (i.e., at unit extension-time) if U is dynamic;
- when unit U is defined (i.e., at unit compile-time) if U is static (see Section 2.4).

Bindings for lazy goals, instead, can be solved only later, when they are called.

5.1. Memory Management in S-WAM

The four dynamic areas of WAM [33] maintain their original meaning in S-WAM:

(1) the local stack stores backtracking information and the procedure environments;
(2) the global stack stores data structures created during the unification process;
(3) the trail stack stores the address of variables which have to be unbound during backtracking;
(4) the code area stores the code of the program being executed.

Local, global and trail stacks grow during the computation and may shrink when backtracking occurs. Local stack may also diminish when Tail Recursion Optimization [5] or trimming [33] is performed.

The first issue to face in designing S-WAM is how to represent partial and global contexts and units belonging to them. From the operational semantics (see Section 4) it immediately follows that the partial context can always be obtained by popping some units from the global one. This property guarantees that an efficient

implementation can be obtained by using only one stack (called the context stack) and two different registers. The first (**GC**, global context register) represents the global context. The second (**PC**, partial context register) the partial context.

To consistently handle backtracking, a register pointing to the top of the stack (**IE_top**) is also added to the register set.

The context stack grows whenever an extension $U \gg G$ (or $U \ggg G$) occurs and shrinks when G is deterministically solved or definitely fails.

The structure of both choice-points and environments of the WAM have been expanded to take the new S-WAM registers into account. In particular, PC, GC and IE_top are saved in choice-points and restored during backtracking.

The problem is now the representation of the units belonging to the context. Let us, from now on, consider only dynamic units. (For static units see Section 5.3.) Notice that you can have multiple occurrences of each unit in the context, i.e., multiple instances of the same unit U. An instance of U is created, in practice, by means of a context extension of $U \gg G$ or $U \ggg G$ kind. When such a creation is required, two different choices might be adopted:

(1) **code copying**: a new private "copy" of U code is made, specialized with respect to the current context;
(2) **code sharing**: several, different instances of U share the same, reentrant code. Each instance of U refers to a different binding environment for predicate calls.

These two alternatives are, respectively, similar to the mechanisms of structure copying and structure sharing in traditional Prolog implementations [5].

Code copying requires on-line compilation and may produce a great amount of code, especially when recursion involving context extension is performed.

In order to obtain more compact use of the memory without code replication for each unit instance, code sharing for units is adopted in S-WAM. This choice corresponds to space-efficient code (i.e., code which uses less memory during execution), and implies only indirect access for eager predicate calls. In this way, each instance of a unit U is associated with a code (shared among all the instances of U) and with a private set of references for eager goals and predicate extensions in U.

These private references are recorded in a new data structure (called *instance environment*) allocated on the context stack whenever a context extension involving U occurs.

Notice that even if an instance environment of a unit U can be logically discarded at the end of goal $U \gg G$ by restoring the old values of PC and GC, it is necessary to physically maintain it on the context stack if G has not been deterministically solved. For this reason the register IE_top is necessary.

With reference to the code representation, in the code area a global table, T, is maintained where, for each unit U, the address of the compiled code of U in the code area is reported.

The compiled code of U consists of the compiled code of procedures defined in U and some additional information. Additional information consists of:

(1) a table for *visible* procedures defined in U ($visible declaration) where for each procedure name the corresponding address is reported;
(2) the type of unit (e.g., dynamic in this case);

(3) a local table (*ET*) maintaining each predicate name for eager predicate calls or extending procedures (i.e., appearing in an $extends declaration) in *U*.

In order to better understand the instance environment structure and function, we have to specify how the S-WAM solves bindings for predicate calls. Local predicate calls of a unit *U* can be statically bound to clauses of *U*. Conversely, definitions for eager and lazy predicate calls are sought in the current partial or global context respectively. Eager bindings depend on the context on which *U* is allocated/nested, and can be determined as soon as the instance is allocated on the context (and at compilation time for static units—see Section 5.3). Thus, the idea is to record these bindings in a proper area of the instance environment, so that they can be directly used for further calls. A similar area has no meaning for lazy calls, since they are to be solved anew each time the call occurs.

Moreover, S-WAM saves in the instance environment all the information needed for restoring the previous context at the end of a context extension. In particular, an instance environment for *U* consists of:

(1) a number of cells, statically determined by the dimension of the *ET* table of *U* (i.e., by the number of eager calls and extending procedures occurring in *U* code), where bindings for eager predicate calls and extending procedures are recorded. The position *i* of a cell in the instance environment of *U* corresponds to the *i*-th predicate name in the local table *ET* of *U*;
(2) a slot where the value of PC is saved;
(3) a slot where the value of GC is saved;
(4) a reference (*unit_ref*) to the code of *U* in the code area;
(5) a slot (*chain*) maintaining a reference to the current context (PC or GC) on which *U* is nested. *Chain* slots maintain the links between instance environments on the context stack;
(6) a slot where the value of IE_top is saved.

In Figure 2, a snapshot of a computation, together with the structure of the instance environment, is reported. Each cell of an instance environment can be

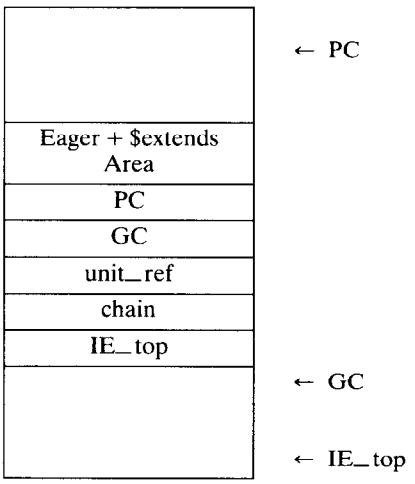

FIGURE 2. A possible state of the context stack.

bound at extension-time. Binding an instance environment cell for predicate p/n requires inspection of the instance environments linked together, starting from the one referred to by PC and searching for some visible code for p/n. If a code is found, its address is stored in the cell. Each cell of an instance environment IE is composed of two slots. The first (referred to as P(IE)) maintains a reference to the code area, i.e., it contains the address of the definition found for p/n. The second (referred to as PC(IE)) refers to the instance environment where the search ended with success. If no code is found, the address of a failing procedure is inserted.

Dealing with contexts and eager/lazy predicate calls in particular suggests considering an address for such calls composed of two information items (⟨program pointer, partial context⟩). In this setting a binding for an eager/lazy predicate call is given by a couple ⟨P, PC⟩, where P is a reference to the code area and PC to the context stack. Since executing an eager or lazy call may modify both these registers, a continuation register, (CPC), saved in the environment, is also introduced for PC.

5.2. *S-WAM Instructions*

To maintain full compatibility with WAM, S-WAM adds few new instructions to the WAM instruction set and leaves unchanged, whenever possible, the behavior of WAM instructions.

5.2.1. Call and Execute Instructions. In both dynamic and static units, local predicate calls are compiled into **call p/n, m** or **execute p/n** instructions, as for standard WAM. Four new instructions are introduced to deal with eager and lazy predicate calls. Given a unit u, eager predicate calls are compiled into one of the following instructions:

call_eager Pi, m
execute_eager Pi

where *Pi* is a symbolic reference which can be solved (in dynamic units) only at context extension-time, i.e., when an instance of u is created by either goal $u \gg G$ or $u \ggg G$. When the i-th cell of the current instance environment (i.e., that referred to by PC) has been bound, both instructions modify the value of the program counter register P and the register PC with the values stored in this cell. Moreover, instruction *call_eager* also saves the previous values of P and PC registers in the corresponding continuations (i.e., CP, CPC).

Lazy predicate calls for a predicate q with arity n are transformed in the following S-WAM instructions:

call_lazy q/n, m
execute_lazy q/n

where the right address for q/n is determined only when executing these instructions. Both instructions search for some visible definition for procedure q/n along the global context. To perform this search, the value of PC is made equal to the value of GC (see the operational semantics in Section 4), and some visible code for q/n is sought in the unit corresponding to the current instance environment (i.e., the one referred to by PC). This search continues recursively along the chain of instance environments until some visible code for q/n is found. If a code is found, it is executed, otherwise backtracking occurs.

Since a procedure address is now seen as a couple ⟨P, PC⟩ and two continuation registers are provided, the **proceed** instruction forces the value of CP and CPC into P and PC respectively.

5.2.2. Indexing Instructions. For each extending procedure *p/n* ($extends declaration) of a unit *u*, standard WAM **try_me_else** instructions are used to allocate a choice point even if the *p/n* is deterministic. The code for the last clause of *p/n* is preceded by the **retry_me_else** instruction and finally followed by the new instruction: **trust_extends Pi**, if *p/n* is the *i*-th predicate in the *ET* table of *u*.

The reference *Pi* is left unsolved at compile-time and will be solved only at context-extension time, as happens for *call_eager* and *execute_eager* instructions. The corresponding values determined by searching along the partial context are inserted in the *i*-th cell of the instance environment of *u*.

trust_extends instruction is very similar to the WAM **trust** the only difference being that P and PC registers are now set to the values stored in the *i*-th cell of the current instance environment (i.e., that referred to by the current PC), in order to perform interunit backtracking.

Example 5.1. Let us consider the following program:

```
unit(eq1):
$visible(equal/2).
equal(X,X).
unit(list2):
member(X,[Y|_]) :- equal(X,Y).
member(X,[_|Z]) :- member(X,Z).
permutation([ ],[ ]).
permutation(L,[X|P]) :- del(X,L,L1),permutation(L1,P).
unit(eq2_II):
$extends(equal/2).
$visible(equal/2).
equal(*,_).
equal(?_).
```

The resulting compiled code is reported in the following:

```
unit eq1
% local equal/2
procedure equal/2 visible
_1528:
        get_value X1,X2
        proceed

unit list2
  %local permutation/2
  %local member/2
  %Eager and Extends Table [del/3,equal/2]
  _4298:
        try_me_else _4314
```

```
    _4326:
            get_list X2
            unify_variable X2
            unify_cdr X8
            execute_eager P2
    _4314:
            trust_me_else fail
    _4474:
            get_list X2
            unify_void 1
            unify_cdr X2
            execute member/2
  procedure permutation/2
            switch_on_term _5582, _5586, _5586
    _5582:
            try_me_else _5618
    _5630:
            get_nil X1
            get_nil X2
            proceed
    _5618:
            trust_me_else fail
    _5586:
            get_variable X3,X1
            get_list X2
            unify_variable X1
            allocate 2
            unify_cdr Y1
            put_value X3,X2
            put_variable Y2,Y3
            call_eager P1,2
            put_unsafe_value Y2,X1
            put_value Y1,X2
            deallocate
            execute permutation/2

unit eq2_II
  % local equal/2
  % Eager and Extends Table [equal/2]
  procedure equal/2 extends visible
            switch_on_term _2870, _2874, _2874
    _2890:
            try_me_else _2906
    _2918:
            get_constant *,X1
            proceed
    _2906:
            retry_me_else _2874
```

```
_2990:
        get_constant ?,X1
        proceed
_2874:
        trust_extends P1
_2870:
        try_me_else _3118
        switch_on_constant 3, _3138
_3138:
        ?
        _2990
        *       tcdr
        _2918
_3118:
        retry_2874

end
```

Standard indexing techniques, taken into account to perform optimized code access, can be straightforwardly extended in order to deal with predicate extension.

5.2.3. Allocation / Deallocation of Instance Environments. Extension goals (i.e., goals of the kind $u \gg G$ or $u \ggg G$) are compiled by using the new instructions:

allocate_c_ctx u
allocate_l_ctx u
deallocate_ctx

Both *allocate_c_ctx u* and *allocate_l_ctx u* allocate an instance environment for u on top of the instance environment stack and, respectively, link it to the current partial and global context. Conversely, *deallocate_ctx* deallocates the current instance environment (i.e., that referred to by GC). The actions performed when executing an *allocate_c(_l)_ctx u* instruction can be summarized as follows:

(1) the address (*ac*) of u code is found in the global table T, if u exists, otherwise the instruction fails;
(2) a number of cells is allocated on top of the instance environment stack in order to store binding for eager predicate calls and extending procedures of u. This number is determined by the dimension of table ET of u;
(3) the values of PC and GC are saved in the instance environment;
(4) *ac* is inserted in slot *unit_ref*;
(5) the value of PC (GC for linear extension) is inserted in the *chain* slot;
(6) finally, the old value of IE_top is saved in the instance environment, and registers IE_top, PC and GC are updated.

The *deallocate_ctx* instruction logically deallocates the instance environment indicated by GC (referred to as *IE*, in the following) and restores the values of PC and GC saved in *IE*. *IE* is physically discarded (i.e., the value of IE_top is set to the old value saved in *IE*) only if no choice-point exists referring to it. In more detail, if register B refers to the last choice-point on the local stack, *IE* is physically

discarded only if (IE_top = GC) and (GC > GC(B)), where GC(B) is the value of GC saved in the last choice-point.

Example 5.2. Let us consider the following units:

unit(main):
go :- eq1 ≫ eq2_II ≫ list2 ≫ member(*, [a, b, c]), write(ok).
unit(list2_II):
$visible(member/2).
member(X, [Y|_]) :- equal(X, Y).
member(X, [_|Z]) :- member(X, Z).
permutation([ ], [ ]).
permutation(L, [X|P]) :- delete ⋙ del(X, L, L1), permutation(L1, P).
The following compiled code is then obtained:

```
unit main
% local go/0
procedure go/0
_1804:
        put_list X2
        unify_constant a
        unify_constant b
        unify_constant c
        unify_nil
        put_constant *, X1
        allocate_ctx eq1
        allocate_ctx eq2_II
        allocate_ctx list2
        allocate 0
        call_lazy member/2, 0
        deallocate_ctx
        deallocate_ctx
        deallocate_ctx
        put_constant ok, X1
        escape write/1
        deallocate
        proceed
unit list2_II
% local permutation/2
% localc member/2
% Eager and Extends Table [equal/2]
procedure member/2 visible
_4310:
        try_me_else _4326
_4338:
        get_list X2
        unify_variable X2
        unify_cdr X8
        execute_eager P1
```

```
_4326:
        trust_me_else fail
_4486:
        get_list X2
        unify_void 1
        unify_cdr X2
        execute member/2
procedure permutation/2
        switch_on_term _5602, _5606, _5606
_5602:
        try_me_else _5638
_5650:
        get_nil X1
        get_nil X2
        proceed
_5638:
        trust_me_else fail
_5606:
        get_variable X3,X1
        get_list X2
        unify_variable X1
        allocate 2
        unity_cdr Y1
        put_value X3,X2
        put_variable Y2,X3
        allocate_1_ctx delete
        call_lazy del/3,2
        deallocate_ctx
        put_unsafe_value Y2,X1
        put_value Y1,X2
        deallocate
        execute permutation/2
end
```

5.3. Compilation of Static Units

For a static unit U, not only local but also eager predicate calls and alternative definitions for extending procedures can be bound at compile-time by performing the same operations (e.g., the search in the associated partial context, represented by the closure of U) as at extension-time for dynamic units. This shows why static units are more efficient than dynamic ones. The price paid is a more complex compilation phase.

In order to avoid the production of a new specialized code, sometimes very large, we choose to maintain for static units the same code-sharing representation adopted for dynamic ones. In more detail, when extending the context with a static unit u_N, if closure $(u_N) = [u_N, u_{N-1}, \ldots, u_1]$, then $N+1$ instance environments (respectively for unit top, $u_1, \ldots, u_N$) are allocated on the context stack. In other

words, a context extension of the kind $u_N \gg g$ is compiled in the following sequence of S-WAM instructions:

```
allocate_c_ctx top
allocate_c_ctx u1
...
allocate_c_ctx uN.
```

Accordingly, N + 1 deallocate operations are performed.

However, in the case of static units, eager calls and extending procedures of a unit U are directly bound at compile-time and recorded in a structure associated with U and called *unit environment scheme*.

The unit environment scheme of a unit U is the static version of the instance environment. In it, the compiler writes the bindings for the eager predicate calls and the extending procedures of U in a symbolic form. In particular, for each binding, it stores the address of the corresponding code found in the code area of a unit U′ which belongs to the closure of U, and an offset which identifies the instance environment of U′ with respect to the instance environment of U. When using U (i.e., performing a context extension operation involving U), S-WAM builds an instance environment for U on the basis of the unit environment scheme of U. In particular, each offset is transformed into a direct reference to an instance environment on the basis of the current value of the register PC.

Example 5.3. Let us consider the following sample program:

```
unit(u1, static( [u2] ) :
$extends(a/0).
a :- b.
unit(u2, static( [u3] ) :
$visible(a/0).
a.
unit(u3) :
$visible(a/0).
$visible(b/0).
b :- #c,#d.
```

We have that closure(u1) = [u1, u2, u3]. Eager predicate calls (b) and alternative definitions for extending procedures (a) occurring in u1 are therefore bound wrt the context [u2, u3].

In this case definitions for b and a exist in unit u3 and u2 respectively. Thus the compilation associates to u1 the eager table ET = [b, a] and the following unit environment scheme, UES, composed of two cells:

P(UES)	PC(UES)
ref. to u3 code for b	offset-N
ref. to u2 code for a	offset-M

where P(UES) maintains the address of the procedure found in the code area and PC(UES) maintains an offset with respect to PC register in the context stack.

The choice adopted here for implementing static units allows easy integration with dynamic units and greatly simplifies the compilation phase. A different solution, producing a new specialized code, is discussed in [19].

5.4. Optimizations

Up to now, different optimizations have been considered with two main aims:

- to reduce the overhead when standard Prolog programs are executed on the S-WAM;
- to enhance structured logic program execution.

In order to reduce the overhead for standard Prolog program execution, the choice-point has been split into two data structures. The first is very similar to the original WAM structure and is allocated on the local stack. The second, called *choice-point extension*, is allocated on the context stack only if context extensions or lazy and eager calls occur.

To improve performances of structured logic program execution, three optimizations have been taken into account. The first regards the binding-time for instance environment cells and the second concerns the possibility of applying *tail recursion optimization* (TRO) on environment variables as it happens in the standard WAM [5], even if the last subgoal of a clause is an extension goal. Finally, a Partial Evaluation scheme has been defined for optimizing structured logic programs.

5.4.1. Choice-Point Splitting. The overhead paid when executing standard Prolog programs on the S-WAM consists of a greater number of memory read and memory write, since both choice-point and environment structures have been expanded to cope with new registers. In particular, a new slot is added to the environment in order to save the CPC register, while four slots have been added to the choice-point to save the values of IE_top, PC, GC and CPC. The greater overhead is due to the four additional memory accesses that occur when executing a *try*, *retry* or *trust* instruction. PC, GC and IE_top values vary in the following cases:

(1) when a context extension occurs (i.e., when an *allocate_c(_l)_ctx* instruction is executed);
(2) when an eager or lazy call is executed (i.e., in correspondence with *call_(execute_)eager* or *call_(execute_)lazy* instructions;
(3) when interunit backtracking is activated, in the case of predicate extension (i.e., when executing the *trust_extends* S-WAM instruction).

None of the previous cases may occur during the execution of standard Prolog programs. For this reason, the choice-point structure has been split into two data structures, the first still allocated on the local stack and the second (called *choice-point extension*) allocated on the context stack. While the value of the IE_top register is still saved in the choice-point on the local stack, a choice-point

extension is allocated on the context stack only if one of the instructions mentioned in 1., 2., and 3. is executed. If this is the case PC, GC and CPC are saved in the new data structure (see Figure 3). As a consequence we now have two different data structures on the context stack, i.e., instance environments and choice-point extensions, in a way similar to what happens on the local stack. Moreover, thanks to choice-point splitting, the test for physical deallocation of the instance environment is now simpler. The result is that an instance environment is physically deallocated when executing a *deallocate_ctx* instruction only if IE_top = GC.

5.4.2. Delaying Binding-Time for Instance Environment Cells. Instance environment cells of a dynamic unit *U* can be bound as soon as they are allocated on the context stack, since their bindings depend only on the context that exists when *U* is allocated. This choice has the drawback that an overhead is also paid at context-extension time for eager predicate calls (or alternative predicate definitions) that will never be executed (or explored). Another possibility could be to leave unbound instance environment cells at context-extension time and determine a binding for them only when the corresponding predicate calls are executed, as happens for lazy predicate calls.

This choice, adopted for default theories of Meta-Prolog [2] implies an overhead that increases with the number of eager predicate calls executed. Such overhead becomes unacceptable if recursion is heavily used.

Thus the best solution is to determine bindings for eager predicate calls the first time calls are performed and then record such bindings in the instance environment cells for subsequent calls. This solution has been straightforwardly obtained in the implementation here presented, by slightly changing the behavior of *allocate_c(l_)ctx, call_(execute_)eager* and *trust_extends* instructions.

5.4.3. TRO on Environment Variables. Given a clause C, if the last subgoal of C is a context extension of kind u ≫ G (or u ⋙ G), in order to correctly perform instance environment allocation and deallocation, the execution control flow has to return to C, after execution of the code for G.

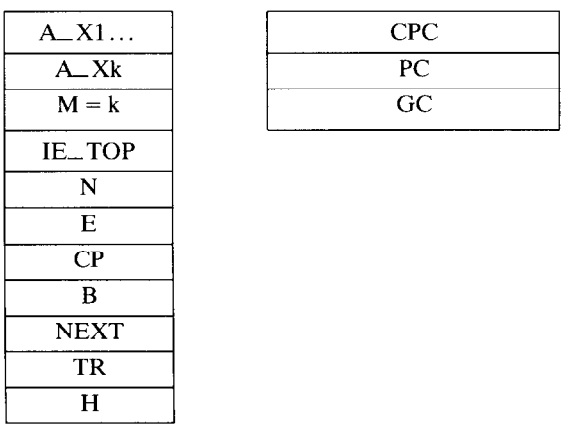

FIGURE 3. Choice-point and choice-point extension.

Example 5.4. Let us consider the following procedure:

p(X) :– g(X), u1 ≫ u2 ≫ f(X).

It is compiled into the following S-WAM code, where a context extension occurs as last subgoal in the clause body:

```
procedure p/1
_3420:
        allocate 1
        put_variable Y1,X1
        call g/1,1
        allocate_c_ctx u1
        allocate_c_ctx u2
        call_lazy f/1,1
        deallocate_ctx
        deallocate_ctx
        deallocate
        proceed
```

Notice that, in the compiled code of procedure p/1, to maintain the values of continuation registers CP and CPC, the clause environment cannot be deallocated until the last subgoal ends, i.e., the "deallocate" instruction is executed just before returning the control to the parent clause ("proceed" instruction). Therefore, TRO [5] on environments allocated on the local stack—peculiar of standard WAM—is no longer applicable. Notice also that we only have to maintain environments to record continuation registers and therefore, correctly handle return. To perform TRO even when we have an extension goal as last subgoal, continuation registers can be saved in another area, and environments can be deallocated before the last call, as it happens in standard WAM. To this purpose, a new instance environment structure has been defined, to record continuation registers and correctly restore them (when executing the "deallocate_ctx" instruction) before the execution of the "proceed" instruction.

An instance environment with the modified structure (see Figure 4) is allocated/deallocated by means of the new S-WAM instructions:

allocate_last_c_ctx u
allocate_last_l_ctx u
deallocate_last_ctx

In this way, TRO is performed on the local stack, but greater instance environments are allocated on the context stack.

Eager + $extends area
CPC
CP
GC
unit_ref
chain
IE_top

FIGURE 4. A different structure for instance environments.

Example 5.5. The procedure of Example 5.4 is compiled into the following S-WAM code:

```
p/1
        allocate 1
        put_variable Y1,X1
        call g/1,1
        deallocate
        allocate_last_c_ctx u1
        allocate_c_ctx u2
        call_lazy f/1,1
        deallocate_ctx
        deallocate_last_ctx
        proceed
```

Notice that now the "deallocate" instruction is performed before the code corresponding to the last subgoal. The new "allocate_last_ctx" instruction is used last to allocate the new instance environment which records the values of continuation registers just restored by means of the "deallocate" instruction. Correspondingly, before the execution of the "proceed" instruction, the new instruction "deallocate_last_ctx" is performed. Notice also that, whenever possible, the "allocate_ctx" and "deallocate_ctx" instructions are used (and therefore, the original, more compact instance environment structure is allocated).

5.4.4. Applying Partial Evaluation. The application of Partial Evaluation [17,20] can be a further enhancement to the compilative approach here presented.

In [6a] we introduce a definition of Partial Evaluation which extends that given in [20] to capture the notions of unit and unit composition and which produces a specialization of the source program with respect to a given goal and a given context in which the goal is to be evaluated. The goal with respect to we perform partial evaluation specifies an initial context and an atomic formula.

The result of the transformation is a new program in which some of (possibly all) the units occurring in the initial context are replaced by their specialized versions. The structure of the source structured logic program is thus maintained in the transformed one, and this ensures full compatibility with the compilation technique presented here. The specialized, transformed program is still a structured logic program running on the S-WAM and taking advantage from the structuring mechanisms, thus maintaining the original readability and avoiding code replication whenever possible.

It is possible to restate in structured logic programming many of the properties which hold for Partial Evaluation in logic programming.

Following the approach introduced in [20], we prove the safeness of the transformation under some sufficient conditions to be checked on the transformed program and the goal. Such conditions actually depend on the different structuring policies we consider.

In the case of static configurations they involve only syntactic checks. As a matter of fact they just correspond to those introduced for standard logic programs in the case of a conservative policy (e.g., in the case of block- and module-based systems), while some further checks on the static structure of the transformed program are required for static evolving systems (e.g., inheritance-based systems).

Thus these two classes of structured programs are actually the best suited for the application of Partial Evaluation [6]. Weaker results can be achieved for dynamically configured systems, since in this case there is hardly a way of statically determining the properties of the transformed program and therefore of the transformation. Some nice properties can still be established provided that we impose some limitations on the ways units can be collected into contexts (for details see [6a]).

5.5. *Dealing with Metaextension Goals and Parametric Units*

Two interesting extensions to the implementation presented so far are discussed in this section. In particular, we take into account the handling of metaextension goals, where the unit name is a variable and thus it is unknown at compile-time, and the implementation of parametric units.

5.5.1. Metaextension Goals. Dealing with extension goals of kind $U \gg g(U \ggg g)$, where the name of the unit is represented by a variable at compile-time, does not require a significant extension to the implementation discussed so far. The addition of a single register (*Unit*), which can occur as argument of *get*, *put* and *unify* instructions, is in fact sufficient to deal with this case. Such a register can be used as argument of the *allocate_c(_l)_ctx* and *allocate_last_c(_l)_ctx* instructions. The only requirement is that this register has already been bound to a constant value at unit extension time.

Example 5.6. Let us consider the following procedure:

```
p(U,X) :- U ≫ q(X).
```

The compilation produces:

```
p/2
        put_value A0,Unit
        get_variable A1,A0
        allocate_last_c_ctx
        call q/1
        deallocate_last_ctx
        proceed
```

Up to now, S-WAM is not able to handle the case of goals represented by variables at compile-time. This is a feature we plan to face in the near future.

5.5.2. Parametric Units. Parametric modules are a powerful construct in building complex software systems. In the logic programming area different proposals exist to cope with this feature. An extension goal of the $U \gg G$ kind could be existentially quantified (i.e., $\exists X\ U \gg G$), suggesting that a unit could contain free variables.

This can be accomplished by using a general term rather than just a constant to refer to a unit, encoding the unit name as the main functor and the parameters as arguments.

In [14], [25] and [26] parametric modules or units are introduced. Such proposals are technically not first-order, since module names can have variables as parameters which are acting as predicate or module names.

In the following, we discuss the implementation of first-order parametric units, i.e., parametric modules, where variable parameters are acting as global variables. Such an implementation requires a major extension of S-WAM design. In particular, the instance environment structure is expanded with a number of cells used to allocate the parameters of a unit.

Example 5.7. Let us consider the following program, derived from [14], in a structured logic programming framework:

```
unit(sort(T)):
$visible(ordered/1).
ordered([ ]).
ordered([X,Y|Z]) :- T = int, arith ≫ less_than(X,Y), ordered([Y|Z]).
ordered([X,Y|Z]) :- T = char, alpha ≫ less_than(X,Y), ordered([Y|Z]).
unit(arith):
$visible(less_than/2).
less_than(0,X) :- natural(X).
less_than(s(X),Y) :- natural(X), natural(Y), less_than(X,Y).
...
unit(alpha):
$visible(less_than/2).
less_than(a,b).
less_than(b,c).
...
```

Unit *sort* is parametric with respect to an individual variable T and contains a definition for predicate *ordered*. According to the value of T, this predicate tests if the corresponding argument list is ordered w.r.t. the right ordering relation. Therefore, the top goal *sort(int)* ≫ *ordered([0, s(s(0))])* succeeds, while *sort(int)* ≫ *ordered([a, b, c])* fails.

Let us note that variables occurring in unit headings are to be considered *global* variables for the clauses of the units. This implies that a different classification for clause variables has to be provided.

Let us consider a clause C of a unit U. A variable V occurring in C is said to be *global* if it also occurs in the U heading. Otherwise, the usual classification as *temporary* or *permanent* variables [33] is followed.

When extending the context with a parametric unit, from the implementation point of view, a proper area is allocated and reserved for storing and maintaining global variables. To this purpose, the instance environment structure is extended with such area, and *allocate(_last)_c(_l)ctx / deallocate(_last)_ctx* instructions have to be modified accordingly. In particular, a new argument is added to allocate_ctx instructions, specifying the number n of parameters of the unit being allocated.

Now an instance environment consists of a vector of value cells for the global variables (G1,...,Gn) of the corresponding unit, together with the area for recording bindings of eager predicate calls and extending procedures and the usual informations saved in it (see Section 5.1). A new set of argument registers (called A_G registers) is added to the S-WAM register set. These registers are unified

with the arguments of a parametric unit before context extension. Accordingly, *get*, *put* and *unify* instructions are modified to handle these new registers.

In order to avoid dangling references, when unifying value cells of the local and context stack, the usual rules which hold for local and global stack value cells are considered. Moreover, to correctly perform backtracking in the trail area, references for global variables that have been bound during unification and that must be unbound on backtracking are also recorded. In particular, when bounding a global variable Gi, its address is recorded in the trail stack if:

address(Gi) > IE_top(B),

where IE_top(B) is the value of IE_top register saved in the last choice-point.

Some compilation examples of parametric units, including the one of Example 5.7, are reported in [30].

Example 5.8. The goal :– *sort(int)* ≫ *ordered([0, s(s(0))])* is compiled into the following sequence of instructions:

```
put_constant int, A_G1
allocate_c_ctx sort, 1
get_variable G1,A_G1
put_list A_X1
unify_constant 0
unity_variable A_X2
unify_nil
get_structure s/1,A_X2
unify_variable A_X2
unify_nil
get_structure s/1,A_X2
unify_constant 0
unify_nil
call_lazy ordered/1, k
deallocate_ctx
```

Notice that, before the context extension involving "sort" (allocate_c_ctx instruction), the constant value "int" is copied in the A_G1 register. To allocate the instance environment, the instruction "allocate_c_ctx sort, 1" is executed. Notice that an additional argument is present to specify the number of "G" cell to be allocated in the instance environment. After the instance environment allocation, the cell G1 is unified with A_G1 register (instruction "get_variable G1,A_G1").

6. S-WAM INSTRUCTION EXECUTION: PERFORMANCE RESULTS

In order to test the performances of the S-WAM, an emulation environment for it has been developed on the basis of a similar environment for standard WAM. The S-WAM environment is based on a special-purpose VLSI microcoded architecture dedicated to execution of S-WAM instructions [9]. This dedicated architecture (called S-PROXIMA, i.e., Structured PROlog eXecutIon MAchine) has been obtained as a rather natural extension of the PROXIMA Prolog machine [8], which directly implements the basic WAM in hardware.

The overall structure of the emulation environment is sketched in Figure 5.

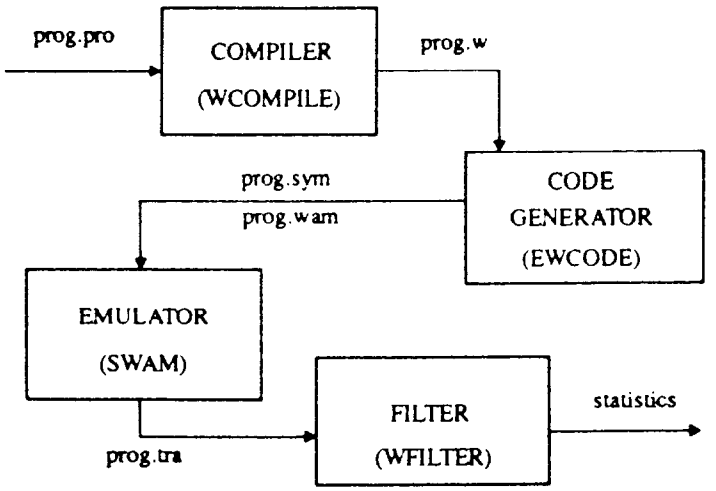

FIGURE 5. The emulation environment.

The compiler has been obtained by extending a standard Prolog compiler [32], written in Prolog, which translates Prolog programs into WAM instructions.

The extensions of the compiler deal with units, context extension operators, bindings for predicate calls with respect to the global or partial context, and interunit nondeterminism. The output of the compiler is a text representation of the resulting S-WAM instructions. The byte code generator, written in Pascal, translates S-WAM instructions into the op-codes directly executable by the processor. In particular, it produces a global table where, for each unit, the address of the corresponding compiled code in the code area is reported.

The S-WAM emulator executes the programs, on the basis of the S-PROXIMA architecture and saves the traces of the operations on a history file. Filtering programs are used to obtain information about program execution statistics. In particular, the filter reports the number of:

- logical inferences;
- S-WAM instructions;
- microcode instructions (called S-WAM0 instructions);
- memory area accesses to the code area, local, global, trail and context stacks);
- memory accesses per Logical Inference.

More details on the resulting system can be found in [9].

6.1. Performance Results

The emulation environment has been used to get information on program execution performances in order to evaluate and compare the overhead due to new language constructs. We use the naive reverse, to reverse a list of length N, as sample program.

The first two tests have been carried out to evaluate the overhead of standard Prolog programs, when executed on the S-WAM. In particular, test 1 reports the

TABLE 1. Performance results

Test	Clock_Cycle	KLIPS
T1N5	735	286
T1N10	2307	286
T1N15	4753	286
T1N20	8086	286
T2N5	794	265
T2N10	2368	279
T2N15	4808	283
T2N20	8114	285
T3UN1	918	229
T3UN2	960	219
T3UN4	1045	201
T3UN8	1153	182
T3UN16	1214	173
T4UN1	1080	195
T4UN2	1188	177
T4UN4	1406	150
T4UN8	1842	114
T4UN16	2714	78

results of the naive reverse execution on the WAM, while test 2 reports the results of the naive reverse, written in a single unit, when executed on the S-WAM.

The tests have been executed for N = 5, N = 10, N = 15 and N = 20. As shown in Table 1, the decrease of performance is limited and less than 3%. For more complex Prolog programs (such as the Cabbage), the decrease of performance is limited but significant (3–9%). For this reason in [9] we analyze the causes of the computational overhead introduced in the S-WAM implementation on S-PROXIMA when standard Prolog programs are executed. In particular, memory accesses do not influence the overall performance. This is a good result from the architectural point of view, since the memory bandwidth is unaffected. This result is even more important, since memory requirements represent a bottleneck of Prolog execution. Therefore, the major overhead of Prolog programs on S-PROXIMA is due to internal activities (register transfers, increments, decrements, tests, etc.) and could be reduced by architectural optimizations.

The aim of Tests 3 and 4 is, instead, to show how performance decreases when a more extensive use of the new language primitives is made. In particular, they show the overhead paid when an eager/lazy predicate call occurs in a variable length partial/global context. The sample programs are the following:

Test 3.UN: Binding eager predicate calls

unit(u0):

go :– app ≫ u0 ≫ ... ≫ nrev ≫ nrev([1,2,3,4,5],X), write(X), fail.

unit(nrev):

nrev([ ],[ ]).

nrev([H|T],Y) :– nrev(T,Z),append(Z,[H],Y).

unit(app):

append([ ],X,X).

append([H|T],Y,[H|Z]) :– append(T,Y,Z).

Test 4.UN: Binding lazy predicate calls

unit(u0):

go:– app ≫ u0 ≫ ... ≫ nrev ≫ nrev([1,2,3,4,5],X), write(X), fail.

unit(nrev):

nrev([ ],[ ]).

nrev([H|T],Y) :– nrev(T,Z),#append(Z,[H],Y).

unit(app):

append([ ],X,X).

append([H|T],Y,[H|Z]) :– append(T,Y,Z).

The search for the eager/lazy call of append/3 is performed in a variable-length context, where UN instances of the same unit, u0, not contributing to the call of append/3 are allocated. The tests have been performed for UN = 1,2,4,8,16.

As regards Test 3, a performance decrease equal to 4.4% occurs when considering two instances of u0 instead of one. When considering four (respectively eight, sixteen) instances of u0 with respect to a single instance of u0, performance decrease is equal to 12.3% (respectively 24.4%, 45%).

As regards Test 4, a performance decrease equal to 9.2% occurs when considering two instances of u0 instead of one. When considering four (respectively eight, sixteen) instances of u0 with respect to a single instance of u0, performance decrease is equal to 23% (respectively 42%, 60%).

These results show that the overhead increases linearly with respect to the context increasing if the policy of binding is eager, while these results cannot be extended to a lazy binding policy. Obviously, this is due to the fact that the search for the binding of eager calls is performed only once, while search for lazy bindings is performed at each call.

We have executed more significant benchmarks [9] to compare what happens when the context handling is heavily performed, i.e., when the new S-WAM instructions are extensively executed, with respect to standard Prolog execution. In particular, only dynamically configured units have been taken into account, since in this case the greater overhead due to contexts is payed.

A significant increase of memory accesses is present when dealing with contexts. The number of the internal transfers increases by about 50% because of the new registers used in the context handling. This is partially due to the noncomplete optimization of the S-PROXIMA architecture for the new S-WAM instructions. Similarly, the number of the arithmetic operations doubles, pointing out the weight of the context stack management and the unit descriptor accesses.

The increase of memory accesses, occurring when dealing with contexts, is mainly due to the management of the new memory structures. The trail and the global stack are unaffected, while the local and the context stacks together with the Unit Table are more intensively used. In the future we plan to revise the S-PROXIMA architecture on the basis of the obtained measurements for structured logic programs to optimize S-WAM instruction execution and to obtain higher performance also when contexts are deeply used.

We have also considered a different technique for implementing structured logic programming based on a straightforward automatic translation of structured logic programs in standard Prolog ones that can be executed on the WAM without extensions.

As can be inferred from the operational semantics, in this case we have to maintain new data structures (i.e., Prolog lists) for recording the contexts (GC and PC). The major overhead of this solution is due to the fact that in the most general case, we perform a more complex unification phase, since we add two contexts as extra-arguments to each predicate definition and predicate call involved.

The basic difference of our approach with respect to the naive translation is that in the S-WAM contexts are not explicitly represented as extra-arguments, but are maintained in an internal data structure (i.e., the context stack) implicitly handled by the machine.

When executing structured logic programs "via" naive translation on a standard WAM, the performance decreases with respect to a direct compilation on the S-WAM. This is mainly due to the greater number of unifications which have to be performed when following the first implementation approach. As a result, we have a greater number of access to the global area. In fact, the implementation of structured logic programs on the S-WAM turned out to be much more efficient (a order of magnitude in terms of KLIPS) with respect to an implementation by naive translation. In particular, in some significant examples [30], the performance decreases from about 230 KLIPS to about 50 KLIPS when using naive translation instead of direct compilation on the S-WAM.

7. RELATED WORKS

Many proposals exist that extend logic programming towards structuring mechanisms (see Figure 1). A detailed discussion of these proposals and their comparison with the one presented here in terms of expressive power, flexibility and declarative semantics is beyond the scope of this paper and can be found in [3]. Here we focus on implementation issues.

Most proposals for structuring logic programs [15, 26] face the implementation issue by translating, if possible, the structured program into a Prolog program that can be executed in a standard WAM.

As pointed out in Section 6.1, the main problem of this approach is the overhead introduced into the transformed Prolog program, due mainly to a greater number of unifications in clause heads. For this reason, the definition of an extended WAM directly supporting powerful structuring mechanisms, as described in this paper, seems to be a more efficient and satisfactory solution.

Also Bacha's implementation of Meta-Prolog [2, 31] follows this approach, even though some differences exist. Some differences are related to the basic mechanisms provided for unit definition and combination and can be summarized as follows:

(1) in Meta-Prolog both permanent and temporary theories exist. Permanent theories are always present in the system and can be accessed via their names, while temporary theories are without names and are only accessible in the environments where the variables bound to their internal representation exist. In our proposal, units can be assimilated to permanent theories while temporary theories are not provided;
(2) in Meta-Prolog, the concept of dynamic context extension does not exist. A limited form of dynamic combination of theories in Meta-Prolog can be obtained by using virtual theories.

(3) in Meta-Prolog, the evolving policy is not provided.

With respect to the implementation, the following differences between our proposal and Meta-Prolog can be pointed out:

(1) in Meta-Prolog the code area is eliminated: the code of theories is in the heap area. We do not adopt this choice since we have no temporary theories;
(2) we adopt an explicit representation of the context as a set of unit instances. The code of each unit U is composed of the procedures explicitly defined in U. Access to the right code for the procedures called but not defined in U is performed by searching in the current context. Meta-Prolog, instead, represents all the procedures for the same predicate p/n in a single data structure $R(p/n)$ independently of the unit they belong to;
(3) while in Meta-Prolog each goal p/n is solved at execution-time, (i.e., when the goal is invoked, by searching for the right code in $R(p/n)$), in our proposal this happens only for lazy goals. The right code for local and eager goals is found, in fact, at compile- and extension-time, respectively;
(4) the difference in code representation determines two different searching algorithms to find the right code for a predicate call p/n. In our proposal the search takes place along a single branch of the tree, i.e., along the current context, as happens for the algorithm described in [31]. While in our case the right branch is straightforwardly determined by the current context, in Meta-Prolog the branch has to be determined by the algorithm in a more complex way. However, this overhead is balanced in Meta-Prolog by the fact that only the theories where p/n is defined are taken into account.

8. CONCLUSIONS

This paper proves that expressive and flexible mechanisms for structuring logic programs can be efficiently implemented by easily extending well-known compilation techniques used for standard logic programming.

The implementation described is based on an extension of the Warren Abstract Machine (S-WAM), where a new stack, new registers and new instructions are added.

Performance results, obtained by using an emulation environment, based on a specialized microprogrammed architecture, show that standard Prolog programs pay a limited overhead when executed on S-WAM. Moreover, directly extending the WAM proved to be a more efficient and satisfactory approach with respect to a preprocessing in Prolog code.

With reference to efficiency increasing, we define a partial evaluation scheme for structured logic programs. Since this scheme preserves the original modular structure of the programs, it ensures full compatibility with S-WAM and the compilation technique addressed here. Significant applications already exist that use structured logic programming. In particular, an advanced logic programming environment has been implemented in the context of the Esprit Project n. 973 (ALPES) [1], taking advantage from the flexibility of the mechanisms here discussed. Moreover, in [18] contexts and units have been used to build an expert

system based on the blackboard model. Finally, contexts turned out to be a suitable tool also for the implementation of negation as inconsistency [10].

Block-, module-, inheritance-based systems, together with systems for hypothetical reasoning, can be implemented and integrated on this efficient architecture [3]. Nevertheless, up to now, no tool has been provided to translate different languages based on these structuring concepts into the general framework here presented. We will overcome this limitation in the near future.

We are indebted to Luis Monteiro and Antonio Porto, since this work has been possible because of their original ideas on Contextual Logic Programming. We thank Pierluigi Civera, Gianluca Piccinini and Maurizio Zamboni who have worked with us in implementing the emulation environment and testing S-PROXIMA architecture. We are also grateful to Antonio Brogi who contributed to the definition of the structuring framework and to Michele Bugliesi who contributed to the partial evaluation scheme. We also thank the anonymous referees for their helpful comments.

REFERENCES

1. ALPES Consortium, ALPES Final Technical Report, 1989.
2. Bacha, H., Meta-Prolog Design and Implementation, in: *Proceedings of the 5th International Conference and Symposium on Logic Programming*, Seattle, Wash., 1988.
3. Brogi, A., Lamma, E., and Mello P., A General Framework for Structuring Logic Programs, C.N.R. Technical Report, "Progetto Finalizzato Sistemi informatici e Calcolo paralello," 1990.
4. Brogi, A., Lamma, E., and Mello, P., *Inheritance and Hypothetical Reasoning in Logic Programming*, Pitman Publishing, Stockholm, 1990.
5. Bruynooghe, M., The Memory Management of Prolog Implementation, in: Clark and Tarnlund (eds.), Logic Programming, Academic Press, San Diego, Calif., pp. 83–98, 1982.
6. Bugliesi, M., Lamma, E., and Mello, P., Partial Evaluation for Hierarchies of Logic Theories, in *Proceedings of the 1990 North American Conference on Logic Programming*, 1990.

6a. Bugliesi, M., Lamma, E., and Mello, P., Partial Deduction for Structured Logic Programming, C.N.R. Technical Report, "Progetto Finalizzato Sistemi Informatici e Calcolo Parallelo," N. 4/7, August 1990 (also to appear in Journal of Logic Programming).

7. Cavalieri, M., Lamma, E., and Mello, P., An extended Prolog Machine for Dynamic Context Handling, in: *Proceedings ECAI-88*, Munich, 1988.
8. Civera, P., Piccinini, G., and Zamboni, M., Implementation Studies for a VLSI Prolog Coprocessor, IEEE MICRO, 1989.
9. Civera, P., Lamma, E., Mello, P., Natali, A., Piccinini, G., and Zamboni, M., Implementing Structured Logic Programs on a Dedicated VLSI Coprocessor, in: *Proceedings of the Workshop on VLSI for Artificial Intelligence and Neural Networks*, Oxford, England, 1990.
10. Demo, B., Negation as Inconsistency and Structured Logic Programming, in: *Proceedings of the 2nd Japan-Italy-Sweden Workshop on Parallel Processing and Logic Programming*, Stockholm, Sweden, 1990.
11. Fukunaga, K., and Hirose, S., An Experience with a Prolog-Based Object-Oriented Language, *OOPSLA-86*, Portland, Oreg., 1986.
12. Gabbay, D. M., and Reyle N., N-Prolog: An Extension of Prolog with Hypothetical Implications, in: *J. Logic Programming*, 4:319–355 (1984).

13. Gallaire H., Merging Objects and Logic Programming: Relational Semantics, in: Proceedings AAAI'86, Philadelphia, PA, 1986.
14. Giordano, L., Martelli, A., and Rossi, G. F., Extending Horn Clause Logic with Module Constructs, DI University of Torino Technical Report, Turin, Italy, 1989.
15. Giordano, L., Martelli, A., and Rossi, G. F., Local Definitions with Static Scope Rules in Logic Languages, in: *Proceedings of the FGCS International Conference*, Tokyo, 1988.
16. Kauffman, H., and Grumbach, A., MULTILOG: MULTIple worlds in LOGic Programming, in: *Proceedings ECAI-86*, Brighton, (U.K.), 1986.
17. Komorowski, H. J., A Specification of an Abstract Prolog Machine and Its Application to Partial Evaluation, Ph.D. Dissertation, Linkoping University, 1981.
18. Lamma, E., and Mello, P., A Knowledge-Based Assistant for Real-Time Planning and Recovery in Automatic Train Protection Systems, in: *Proccedings of the Reliability on the Move SARSS'89*, Bath, U.K., 1989.
19. Lamma, E., Mello, P., and Natali, A., The Design of an Abstract Machine for Efficient Implementation of Contexts in Logic Programming, in: *Proceedings of the 6th ICLP*, Lisbon, Portugal, 1989.
20. Lloyd, J. W., and Shepherdson, J. C., Partial Evaluation in Logic Programming. Technical Report CS-87-09, University of Bristol, U.K., 1987.
21. McCarty, L. T., Clausal Intuitionistic Logic. 1. Fixed-Point semantics, in: *J. Logic Programming* 5:1–31 (1988).
22. McCabe, F. G., Logic and Objects: Language, Application and Implementation, Ph.D. Dissertation, University of London, 1988.
23. Mello, P., Inheritance as Combination of Horn Clause Theories, in: M. Lenzerini, D. Nardi, M. Simi, (eds.), *Inheritance Hierarchies in Knowledge Representation*, Wiley, 1990.
24. Mello, P., Natali, A., and Ruggieri, C., Logic Programming in a Software Engineering Perspective, in: *North American Conference on Logic Programming*, Cleveland, Ohio, 1989.
25. Miller, D., A Theory of Modules for Logic Programming, in: *Proceedings of the 1986 Symposium on Logic Programming*, Salt Lake City, (Utah), 1986.
26. Monteiro, L., and Porto, A., Contextual Logic Programming, in: *Proceedings of the 6th ICLP*, Lisbon, Portugal, 1989.
27. *MProlog Language Reference Manual*, LOGICWARE, Inc., Toronto, Canada, 1985.
28. Nakashima, H., and Nakajima, K., Hardware Architecture of Sequential Inference Machine PSI-II, Technical Report TR-265, ICOT, Tokeyo, J, 1987.
29. Nakashima, K., Knowledge Representation in Prolog/KR, in: *Proceedings of the International Symposium on Logic Programming*, Atlantic City, N.J., 1984.
30. Pagnoni, V., Implementing Generic Modules in Logic Programming, B.A. Thesis, University of Bologna, Italy, 1990.
31. Bacha, H., Meta-level Programming: A Compiled Approach, in: *Proceedings of the 4th ICLP*, Melbourne, Australia, The MIT Press, 1987.
32. Van Roy, P., A Prolog Compiler for the PLM, Report No. UCB/CSD 84/203, Computer Science Division, University of California, Berkeley, 1984.
33. Warren, D. H. D., An Abstract Prolog Instruction Set, SRI Technical Note 309, SRI International, Menlo Park, Calif., 1983.
34. Warren, D. S., Database updates in Prolog, in: *Proceedings of the FGCS International Conference*, Tokyo, 1984.
35. Wegner, P., and Szonik, S. B., Inheritance as an Incremental Modification Mechanism or What Like Is and Isn't Like, in: *Proceedings of ECOOP'88 and LNCS*, Oslo, N, 1988.
36. Zaniolo, C., Object-oriented Programming in Prolog, in: *Proceedings of the International Symposium on Logic Programming*, Atlantic City, N.J., 1984.