An Architecture for Prolog Extensions

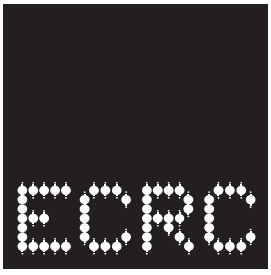
Micha Meier
Joachim Schimpf

ECRC-ECRC-95-06

technical report ECRC-ECRC-95-06

An Architecture for Prolog Extensions

Micha Meier
Joachim Schimpf



European Computer-Industry Research Centre GmbH (Forschungszentrum)
Arabellastrasse 17
D-81925 Munich
Germany
Tel. +49 89 9 26 99-0
Fax. +49 89 9 26 99-170
Tlx. 52 69 10

For more information please contact : michaecrc.de, joachim@ecrc.de

Abstract

We address the task of an efficient implementation of Prolog extensions. Prolog is a very good language for prototyping and almost any extension can be quickly written in Prolog using a Prolog interpreter and tested on small examples. It is harder to find out if the results scale up to large, real life problems, though. A Prolog interpreter, even if partially evaluated with respect to a given problem, quickly hits the space and time limitations and so more elaborate approaches to the implementation are necessary. In this article we describe an architecture of a Prolog system that gives the user enough support to quickly prototype new extensions and at the same time to implement them efficiently and incrementally. This architecture has been used to build the **SEPIA** and **ECL^{i}PS^e** systems.

1 Introduction

Prolog is a very good language for prototyping and for implementation of various new extensions of logic programming. Its declarative semantics, ability to handle program as data and interactive processing give the user enough flexibility to quickly test new ideas. If, however, the researchers want to test the applicability of the extension to large, real-life problems, more elaborate approaches to the implementation have to be taken. Some of them require more work and are more efficient, others have other advantages and disadvantages. We can compare them for instance with the following criteria:

- how easy it is to implement the extension, how much of the normal Prolog functionality has to be duplicated
- the execution speed
- the space usage
- how easily can two or more extensions be combined together.

The most obvious methods to implement extensions are then the following:

1. **Interpreters** can be very quickly written in Prolog, they are simple and easy to understand, often it is even possible to combine several extensions just by combining their interpreters or by slightly modifying them. However, the additional interpretation layer causes a significant overhead and often it is even not possible to use the built-in unification algorithm and thus the whole unification has to be re-implemented in Prolog. Furthermore, the space usage of interpreters is quite high, because all information has to be represented explicitly, it cannot be 'compiled away'. Determinacy detection in interpreted programs is often difficult or impossible and this causes another increase in space usage. The last but not least drawback of this approach is that the whole input program might have to be executed with the interpreter, even if only a small fraction of it requires extended functionalities and the rest is plain Prolog which could be otherwise compiled and executed directly.
2. **Translation by specialising an interpreter**. By partially evaluating a Prolog interpreter with respect to a given program we obtain a translation of the extended program into Prolog. Since this process can be completely automated, it is a very convenient way to obtain a faster implementation without further implementation cost. All advantages of a metainterpreter are kept. This approach may also be more efficient because the additional interpretation layer may be partly removed. The space usage and determinacy detection is better than for a plain interpreter.

This scheme is however usually limited to small and pure programs. In the presence of real-life problems it is often difficult to obtain a significant improvement by the partial evaluation (at least this is what we can say from our experience). Significant extensions also depend largely on dynamic parameters, e.g. data, and cannot be statically transformed into straightforward Prolog programs. For instance, extensions that change the default control rule must explicitly store and process the resolvent if the control depends on dynamic parameters (which it usually does).

3. **Compilation to Prolog**. In this case the extended program is translated into plain Prolog by means of a specialised translator or compiler. Writing an extension compiler is not always an easy task and since its complexity often does not match its efficiency, it is much easier to develop a specialised metainterpreter.

   Combining two extensions in this way may pose some nontrivial problems.

4. **New system from scratch**. This is obviously the most efficient and also the most difficult possibility. Even when one utilizes publicly available Prolog libraries, e.g. the parser, compiler and some built-in predicates, one has to write a huge amount of low level code that has nothing to do with the extension, just to implement a plain Prolog system.

   Currently it makes much more sense to modify an available Prolog system like SICStus or SB-Prolog.

5. **Modifying an existing system** also yields a very efficient implementation, however its cost is still quite high:

   - To modify its functionality, one has first to understand the implementation of the whole system.
   - When modifying the abstract machine, it might be quite difficult to change the compiler to cope with it efficiently. Particularly, we do not consider WAM modifications as a good idea for implementing new extensions. The WAM was an important step in the history of Prolog, but future compilers will have to be based on global analysis and use information for which the WAM is too coarse grained. Taking this into account, extensions through WAM modifications are very complex tasks which lead nowhere.
   - When a new release of the host Prolog system becomes available, the changes for the extension have to be carefully repeated in the new sources, and sometimes the new release may be incompatible with it. The extensions are likely not to gain anything from advanced techniques like global analysis and partial evaluation.
   - As parts of the extension have to be implemented in the low level implementation language and parts in Prolog, it is far too easy to miss the right balance and to put too much emphasis on the low level programming, trying to make the implementation more

efficient. The parts programmed at low level cannot be easily modified or extended and their effect may be outweighed by implementation costs and increased complexity of the system.

- Combining two extensions is extremely difficult and it is usually not even worth trying.

It can be seen that none of these approaches offers the right balance between simplicity and efficiency. It is obvious that efficient implementations have to be *integrated* into the Prolog machine at a low level. However, to make such an integration easy, the Prolog system itself must be built with precisely this goal in mind. Our idea is to define 'sockets' in the Prolog kernel which allow the extensions to be plugged in efficiently. Extended programs are then compiled and run just like the normal ones except that, when some extended feature is required, the control is given through the 'socket' to the extension code, which does the appropriate processing. In this way, the implementation has the simplicity of a metainterpreter and (almost) the efficiency of a modified native Prolog system.

2 What 'Sockets' Are Necessary for an Extension?

It is quite difficult to think of a scheme that would suit every possible Prolog extension. We have tried to cover all areas researched at our institute, ECRC, and our has been shown to be applicable even for other areas, but it is likely that still more sockets could be useful. The processing of a Prolog program can be generally divided into areas like

- reading in the program and storing it in memory,
- control,
- data processing, unification,
- execution of built-in predicates, ...

Every Prolog extension will have to modify at least one of these areas, most extensions will modify several of them. In the following sections we will discuss how the corresponding 'sockets' can be defined to allow efficient and straightforward processing.

2.1 Compilation and Program Storage

First of all, there should be no sockets in the code generator. An optimizing compiler has to have all the knowledge about the compiled code and if at

some point an undefined action could be taken, or unknown instructions generated, the possibility for code improvement would be severely restricted. We believe that we can avoid changes in the compiler by preprocessing the source and by exploiting the event mechanism (see below).

Some extensions use special syntax to denote their objects or their special constructs. Although it is possible to use Prolog operators and compound terms to represent extension objects, this might not always be efficient enough. This problem is solved by introducing *input macros* as a 'socket' in the parser, whose presence in the input triggers an event or simply calling of a specified transformation predicate. In this way, some functors can be reserved to represent special objects and these objects are created by the transformation procedure.

2.2 Built-In Predicates

Every Prolog system contains a number of built-in predicates. Some of them are implemented in Prolog itself, others in the implementation language. When an extension is being connected to Prolog, often it is necessary to modify the behaviour of some of the built-in predicates so that it reflects the extension functionality. Although it is possible to do this by modifying the source code of the predicates, this approach is tedious and error-prone, and usually requires identical or similar changes for many of the predicates. For example, performing arithmetic with the successor function only (i.e. 2 is s(s(0))) would require the modification of all predicates that evaluate arithmetic expressions and inclusion of code that translates between the number and successor representation of the integers. The same would apply if e.g. rational numbers were included in the language.

This problem can be solved in a much more coherent way by including *event handling* in the Prolog engine. Whenever the built-in predicate encounters a situation which is not strictly regular, e.g. a compound term when a number is expected, it raises an event. Depending on the event type, an appropriate event handler is invoked, which will solve the problem, if there is any, and then return to the normal execution. This is equivalent to replacing every built-in predicate with another one that first makes a series of checks to find out whether there are some irregularities present, and if so, a corresponding handler predicate is directly called.

As soon as an event handling scheme is available, it can be used for various other purposes, e.g. to customize the system, to keep control even if things go wrong, etc. The event handling scheme is described in detail in [8].

2.3 Control

2.3.1 Coroutining

The usual left-to-right Prolog selection rule has often to be modified in Prolog extensions. A simple, context-free approach is to specify syntactical conditions which a literal must satisfy in order to be selected. The selection rule proceeds as usual, except that literals which fail to satisfy the selection condition are suspended until the condition is satisfied. This is the so-called *coroutining*, introduced for the first time in Prolog-II [4].

Prolog systems that provide coroutining have often only simple facilities based on Prolog-II's **freeze/2** to suspend a predicate call [3]. For advanced extensions like constraint propagation a more sophisticated design is necessary:

- The user must be able to specify several delaying conditions for one goal, which implies that the goal might be woken by more than one variable. Although this feature can be simulated by **freeze/2**, it has to be provided by the system for efficiency.
- **freeze/2** or declarations as in NU-Prolog [11] are not flexible enough to directly specify complex control strategies. The **delay clauses** introduced in **SEPIA** represent a completely declarative approach to coroutining control – they are meta clauses with a syntax like normal Prolog clauses, e.g.

```
delay p(X, 1) if var(X).
```

The delay clauses allow the specification of even quite complicated control very naturally. For example, the logical conjunction **and(In1, In2, Out)** with its usual definition has to be delayed if both *In1* and *In2* are un-instantiated (otherwise the result is 0 or equal to the other input), and different (otherwise the output is equal to the inputs) and *Out* is not 1 (otherwise both inputs must be 1). Such a condition is naturally expressed with a delay clause as

```
delay and(Op1, Op2, Res) if var(Op1), var(Op2),
                            Op1 \== Op2, Res \== 1.
```

Several delay clauses may be specified for a predicate to allow disjunctive conditions. When executing a delay clause, its head uses pattern matching instead of unification, and no variables in the call may be bound by it.

Another advantage of delay clauses is that they are extensible - although only built-in test predicates are allowed in the body of a delay clause, the user can also add external predicates (written in C) for specialised control. This option

has been preferred to completely general subgoals in delay clauses, because the latter would cause problems in the semantics and in the implementation and we have had no extensions that actually required it.

2.3.2 Manipulation of the Resolvent

For more advanced extensions it is necessary to introduce some context in which a literal can or has to be selected. A general principle that allows processing of the resolvent and selection of the next literal is difficult to implement. Such a mechanism is also very likely to degrade the performance of ordinary Prolog programs and bring them close to metainterpretation. However, a flexible and efficient mechanism can be built on top of coroutining. It is only necessary to allow access to suspended literals, so that the list of all suspended goals can be processed, from which a suspended literal may possibly be selected. Thus e.g. the predicate **suspended_goals(Var, GoalList)** returns the list of all goals suspended due to the specified variable, and **suspended_goals/1** returns all suspended goals.

2.3.3 Occur Check and Other Search Rules

Prolog's lack of the *occur check* and use of *depth-first* search is, for some extensions, a serious problem, because the underlying solver is then neither sound nor complete. Fixing this in an existing system may cause significant overheads as unification with the occur check must be completely coded in Prolog. Therefore, an architecture for Prolog extensions must contain the occur check and alternatives to depth-first search as an option. Using compiler technology, it is possible to have an optional occur check and an optional (e.g. depth-first iterative deepening) search, by specifying different compiler modes so that the compiler either generates the additional instructions (to perform the occur check or to test the execution depth), or not. In this way, we can keep both efficiency and extensibility.

In **SEPIA** we have used this approach to implement depth-bounded search and depth-first iterative deepening search. Predicates compiled in this mode fail if the execution depth exceeds the limit, and a corresponding top-level predicate controls how the limit is set.

2.3.4 Connection to External Systems

Some extensions may need a connection to an external system, usually a non-logical one. The inclusion of such a system into the backtracking scheme of Prolog may pose some problems and new requirements. Our architecture supports external systems with the following features:

- Prolog and C are mutually callable (this is more or less standard in current systems); C functions can also backtrack and suspend.
- Most of the external data can be directly used in Prolog; the number and string formats are compatible and structured data can be mapped on Prolog arrays.
- We provide hooks to notify the external system of
  - failure
  - cut
  - garbage collection of data related to the external system.

The necessity of failure and cut notifications may not be obvious; a typical example of their use is a tight connection of a relational database to Prolog. When a database query starts, it is necessary to open the relation, allocate buffers etc. Since the answer is passed into Prolog by backtracking over all tuples, the relation has to stay open until either all answers have been exhausted and the query fails, or if the execution commits to one tuple with a cut. Using the hook, the database is notified about the cut or failure and it can close the relation and/or remove the write locks on it immediately.

As long as external systems use data in the Prolog format, all garbage can be collected automatically. If this is not possible, the hooks in the garbage collector can be used to define the format of data that can appear on the Prolog heap and all pointers to it.

2.4 Unification

Unification is the core of Prolog and thus many extensions require a modification of the basic unification algorithm, either to include new cases (new data types or an extended notion of unify-ability), or to disallow some existing ones (e.g. unification of non-matching types). When extending unification, it is absolutely crucial to keep the speed of the ordinary, non-extended unification.

Extended unification is therefore provided by a generic data type, called a **metaterm**. A metaterm can be seen as a normal free variable X which has some Prolog term as its associated attribute:

Whenever a metaterm is unified with another one or with a non-variable, an event is raised, and the appropriate event handler takes care of the processing. In **SEPIA** it is the event 10 for the former and 11 for the latter. When a normal free variable is unified with a metaterm, it is directly bound without raising an event, because it carries no internal information that could influence the unification. In this way, Prolog is extended at a conceptually high level,

while keeping the efficiency of a compiled WAM, because normal execution is not at all influenced or slowed down. The operations supported for metaterms are the following:

- Creation, i.e. coupling a variable with an attribute using the predicate

  meta_term(Var, Attr).

- Accessing the attribute, using the same predicate.
- Test if a term is a metaterm using **meta(Term)**.
- Deletion – the attribute is removed and the variable is bound to some term with the **meta_bind(Meta, Term)** predicate, which strips off the attribute from **Meta** and binds it to **Term**.

The metaterms do not represent a separate meta-level of the program – it is rather a data type that supports the amalgamation of the meta and object level. The attribute of the metaterm may be any term, and free variables in it are subject to normal unification.

Metaterms represent a data type with interesting properties which can be used not only for a plain extended unification, but also for a number of other purposes. A metaterm can be seen and processed in several different ways:

- a variable with an attribute
- a variable whose value is known only partially
- a generic data type
- user-definable extension of unification
- a reference data type.

As the last item shows, metaterms can be also used to implement extra-logical and impure features. This, in general, can be an advantage, if the impure features are only used to efficiently implement pure logical mechanisms (like implementing Prolog using C). However, care must be taken not to destroy declarative semantics of the programs.

Metaterms can be used for an efficient and straightforward implementation of various extensions. Below, we list a few of them, although there are certainly many more.

2.4.1 Static Attribute

Sometimes we need to associate some simple and fixed information with a variable, for instance

- the source name of the variable,
- marking quantified variables,
- marking void variables,
- marking selected variables.

This example shows how metaterms can be used to trace successive instantiations of a term:

```
trace_subst(Var) :-
    meta_term(Var, _).      % add a dummy attribute

meta_unify(M, T) :-
    printf("variable %w bound to %w\n", [M, T]),
    % call vars_to_meta recursively for all subterms of T
    apply_to_subterms(vars_to_meta, T),
    meta_bind(M, T).        % do the actual binding

vars_to_meta(X) :-
    meta(X) -> true ; var(X) -> meta_term(X, _) ; true.

% set meta_unify/2 to be called whenever a metaterm is bound
:- set_error_handler(10, meta_unify/2).
:- set_error_handler(11, meta_unify/2).
```

The predicate **trace_subst(Var)** marks the variable with a dummy attribute and whenever such variable is bound, a message is displayed and all variables occurring in the bound term are in turn marked:

```
% An example code:
p(f(X, Y, Z)) :- q(X), r(Y, Z).

q(g(a, U)) :- s(U).

r(Y, Y) :- Y = [a|T], s(T).

s([]).


% The execution trace:
[sepia]: trace_subst(A), p(A).
```

```
variable A bound to f(X, Y, Z)
variable X bound to g(a, U)
variable U bound to []
variable Y bound to Z
variable Z bound to [a|T]
variable T bound to []

A = f(g(a, []), [a], [a])
yes.
```

2.4.2 Dynamic Attribute

A more advanced technique is to give the metaterm an attribute which changes during the execution. This attribute can represent e.g.

- a list of goals that have to be executed when the variable becomes instantiated - "classical coroutining"
- a goal to be executed to obtain or update the value of the variable
- a ψ-term [1]
- implementation of order-sorted logic.

This example generates a lazy list of 1's. The tail of the list is initialised with a metaterm, whose attribute stores the goal **ones/1**. As soon as the list tail is unified, the handler procedure **tail_handler/2** drops the attribute, unifies the list tail and calls the goal **ones/1** which produces a new list element.

```
% Generate a lazy list of ones.
ones([1|X]) :- meta_term(X, ones(X)).

% The handler for metaterm unification
tail_handler(M, T) :-
    meta_term(M, Goal),
    meta_bind(M, T),      % bind the metaterm to T
    call(Goal).           % and execute the goal

:- set_error_handler(10, tail_handler/2).
:- set_error_handler(11, tail_handler/2).

% Example of use:
add_list([X|L], [Y|M], [S|R]) :-
        S is X + Y,
        add_list(L, M, R).
add_list([], _, []) :- !.
```

```
add_list(_, [], []).

[sepia]: ones(Ones), add_list([1, 2, 3], Ones, Result).

Ones = [1, 1, 1|M]
Result = [2, 3, 4]     More? (;)
yes.
```

2.4.3 Generic Data Types

Metaterms can represent a typed variable with run-time type testing. The attribute can store the type name, type value, a list of possible type values or a goal that checks the type when the variable is unified.

Here is an example of a simple type hierarchy which uses built-in type testing predicates.

```
:- op(1150, fx, type), op(700, xfx, :).
type Name = (Type; Rest) :-
    type(Name = Type),
    type(Name = Rest).
type Name = Type :-
    atom(Type),
    assert(type(X, Name) :- type(X, Type)),
    assert(subt(Type, Name)).

Var:Type :-
    meta_term(Var1, Type),
    Var = Var1.

type_type_unify(M1, M2) :-
    meta_term(M1, T1),
    meta_term(M2, T2),
    (subtype(T1, T2) -> meta_bind(M2, M1) ;
    subtype(T2, T1) -> meta_bind(M1, M2)).

type_term_unify(M, Term) :-
    meta_term(M, Type),
    type(Term, Type),
    meta_bind(M, Term).

:- set_error_handler(10, type_type_unify/2).
:- set_error_handler(11, type_term_unify/2).

subtype(T, T).
subtype(T1, T2) :-
    subt(T1, T),
```

```
    subtype(T, T2).

% built-in types
type(I, integer) :- integer(I).
type(F, float) :- real(F).
type(A, atom) :- atom(A).
type(S, string) :- string(S).
type(C, compound) :- compound(C).

% Now define a simple type hierarchy
:- type term = (var; nonvar).
:- type nonvar = (atomic; compound).
:- type atomic = (number; atom; string).
:- type number = (integer; float).

% An example of its use
[eclipse 2]: X:atom, Y:number, X=Y.

no (more) solution.
[eclipse 3]: X:atomic, Y:number, X=Y, X=1.

Y = 1
X = 1     More? (;)
yes.
```

2.4.4 Object-Oriented Programming

If the metaterm is used to represent an object, the metaterm variable represents a reference to the object and the attribute stores the state of the object. If the object changes its state and if it is certain that no references to the old state exist, the state can be modified just by exchanging the attribute of the metaterm.

2.4.5 Constraint Propagation

The attribute of the metaterm stores all constraints imposed on the variable. In combination with the goal suspension (either built-in or also implemented using metaterms), we obtain a powerful tool to implement various kinds of CLP languages, without having to modify the low level code of the Prolog machine. The constraint solver(s) are quite naturally invoked by the event handlers of metaterm unification.

We have been able to implement a package for arithmetic constraints over finite domains of integers and another for (non)equality over arbitrary terms. The whole code is written in Prolog and uses a simple list representation of the domains, but its performance is quite good, it is only 4 – 10 times slower than

the CHIP compiler [5], which is a dedicated CLP system using a bit field representation for domains, and low-level C coding for all constraints processing.

Below are several examples of the definition of simple constraints. The first defines only variables over finite integer domains and their unification.

Defining variables, whose value is from a specified integer finite domain:

```
% Constrain a variable to be from the specified integer interval.
:- coroutine.
:- op(300, xfx, [in, #, <., <..]).
:- import initial/2 from sepia_kernel.

X in [L,H] :-
    (number(X) ->       % only test the bounds
        L =< X, X =< H
    ;
    meta(X) ->          % already constrained
        New in [L, H],
        X = New
    ;                   % constrain the variable
        make_list(L, H, D),   % make a list of integers
        meta_term(X, D)
    ).

make_list(H, H, [H]) :- !.
make_list(L, H, [L|R]) :-
    H > L,
    L1 is L + 1,
    make_list(L1, H, R).

meta_unify(T1, T2) :-
    meta_term(T1, D1) ->
        (meta_term(T2, D2) ->   % both constrained
            intersection(D1, D2, D),
            new_attribute(T1, D1, D),
            meta_bind(T2, T1)
        ;
            memberchk(T2, D1),  % one instantiated
            meta_bind(T1, T2))
    ;
    meta_term(T2, D2) ->
        memberchk(T1, D2),
        meta_bind(T2, T1)
    ;
        T1 = T2.                % both instantiated

:- set_error_handler(10, meta_unify/2).
```

```
:- set_error_handler(11, meta_unify/2).

% Common handling of constants and constrained vars
attribute(A, DA) :-
    meta(A) -> meta_term(A, DA) ; DA = [A].

% Give the variable a new attribute
new_attribute(_, OldD, OldD) :- !.     % no change
new_attribute(Var, _, [Val]) :-        % instantiate it
    !,
    meta_bind(Var, Val).
new_attribute(Var, _, [V|R]) :-        % new domain most be non-empty
    meta_term(New, [V|R]),
    meta_bind(Var, New).               % modify the attribute

domain_range([Min|R], Min, Max) :-
    last_element(Min, R, Max).

last_element(Max, [], Max) :- !.
last_element(_, [E|R], Max) :-
    last_element(E, R, Max).

% Print the variable together with its attribute.
portray(Stream, Var) :-
    meta(Var),
    meta_term(Var, Domain),
    printf(Stream, "<%w%Pw>", [Var, Domain]).

% Example:
[eclipse 2]: X in [1,10], Y in [5,15], X=Y.

X = New<[5, 6, 7, 8, 9, 10]>
Y = New<[5, 6, 7, 8, 9, 10]>
yes.
[eclipse 3]: X in [1, 6], X in [6, 8].

X = 6
yes.
```

The next example adds to the functionality of the previous one the inequality constraint over integers. The inequality must be delayed until one argument is ground as no pruning can be done before this, but then it is completely solved (*forward checking* in the terms of [6]). Note that the control could be extended to delay also in the case when one argument is a plain free variable.

Inequality over integer finite domains.

```
% Delay #/2 until one argument is a non-variable or
% both are equal.
```

```
delay A # B if var(A), var(B), A \== B.
A # B :-
    nonvar(A) -> delete_value(A, B) ;
    nonvar(B) -> delete_value(B, A).

delete_value(N, Var) :-
    attribute(Var, D),
    delete(N, D, ND) -> new_attribute(Var, D, ND) ; true.

% Example of use:
[eclipse 4]: X in [1,5], Y in [3,7], X # Y, X = 4.

Y = New<[3, 5, 6, 7]>
X = 4
yes.
[eclipse 5]: X#Y, X=Y.

no (more) solution.
```

The last example shows a more advanced constraint type – inequality $< /2$ of two integer domain variables. This constraint prunes those elements from the domains of the two variables which are incompatible with the constraint, but then, unless it is trivially satisfied or falsified, it has to wait to be woken as soon as one of the domains is updated, but not necessarily reduced to a single element (this is termed *partial lookahead* in [6]). Note that, without sophisticated control primitives, defining such a predicate would be quite difficult.

The $< /2$ relation for two domain variables.

```
A < B :-
        A <. B.

A <. B :-
        attribute(A, DA),
        attribute(B, DB),
        domain_range(DA, MinA, MaxA),
        domain_range(DB, MinB, MaxB),
        (MinB > MaxA ->
            true                        % solved
        ;
            remove_greatereq(DA, MaxB, NewDA),
            new_attribute(A, DA, NewDA),
            attribute(B, DB1),
            remove_smallereq(DB1, MinA, NewDB),
            new_attribute(B, DB, NewDB),
            (MinB > MinA,
             MaxB > MaxA ->
                A <.. B                         % nothing done
```

```
            ;
                A <. B                  % repeat
            )
        ).

% initial/2 succeeds if (A, B) contains more than one variable,
% but it lets the goal continue as soon as A or B is updated,
% e.g. by modifying their attribute.
delay A <.. B if initial(1, (A, B)).
A <.. B :-
        A <. B.

remove_smallereq([X|Rest], Min, L) :-
    X =< Min,
    !,
    remove_smallereq(Rest, Min, L).
remove_smallereq(L, _, L).

remove_greatereq([X|Rest], Max, [X|L]) :-
    Max > X,
    !,
    remove_greatereq(Rest, Max, L).
remove_greatereq(_, _, []).

% Example:
[eclipse 6]: X in [1, 5], Y in [2, 10],
        Y < X, printf("X = %Pw, Y = %Pw\n", [X, Y]),
        X # 5, printf("X = %Pw, Y = %Pw\n", [X, Y]),
        Y # 2.
X = <New[3, 4, 5]>, Y = <New[2, 3, 4]>
X = <New[3, 4]>, Y = <New[2, 3]>

X = 4
Y = 3
yes.
```

2.4.6 True Metaterms

A term which contains metaterms can be seen as a potential representation of many other terms (instances) and, depending on the context where it is used, it is interpreted as the appropriate one. This means that the term represents an amalgamation of the object level with a meta level and this amalgamation is quite an efficient one.

A typical example of this metaterm interpretation is the representation of the object program in a meta program. Object variables cannot be represented by free variables because this would cause conflicts with the meta variables. Usually this is solved by representing the object variables by ground terms (e.g.

using **numbervars/3**), so that they can be easily recognised at the meta level. In this way, however, an object goal must be either fully interpreted by the meta program, or the goal has to be first reflected to the object level, i.e. the grounded variables transformed again to free ones, and then the goal can be directly executed. Moreover, to obtain the answer substitution, an explicit list of all variables and their ground representations must be created.

When metaterms are available, they can be used to represent object variables, or even any terms, and, the term containing them is interpreted differently depending on the level at which it is being processed.

In the following example a goal is executed directly. However, none of its variables is being bound – a list of substitutions is returned instead. This is achieved by replacing every variable by a metaterm whose attribute is a free variable, and performing all unification on this attribute. When the goal succeeds, the substitution list is simply created from all the attributes.

Execution without variable binding.

```
% Call a goal and return a list of substitutions.
call_subst(Goal, SubstList) :-
    % collect the vars from Goal replaced by metaterms into Vars
    sumnodes(vars, Goal, Vars, []),
    call(Goal),
    var_subst(Vars, SubstList).

% This procedure is applied recursively to every argument of Goal
vars(X) -->
    {meta(X)} -> [] ;    % already meta, ignore
    {var(X)} -> {meta_term(X, _)},
                [X] ;    % add the attribute and put into the list
    [].

% The unification handler. When a metaterm is unified,
%  we unify its attribute instead.
meta_unify(M, Term) :-
    meta_term(M, Term).

:- set_error_handler(10, meta_unify/2).
:- set_error_handler(11, meta_unify/2).

% Build the list of substitutions.
var_subst([], []).
var_subst([Var|Rest], SubstList) :-
    meta_term(Var, Term),
    (var(Term), not meta(Term) ->
        SubstNew = SubstList
    ;
        SubstList = [Var/Term|SubstNew]
```

```
    ),
    var_subst(Rest, SubstNew).

% Examples:
[eclipse 4]: call_subst(f(D, A, g(B)) = f(a, B, C), List).

D = D
A = A
B = B
C = C
List = [D / a, A / B, C / g(B)]
yes.
[eclipse 5]: call_subst(append(A, B, [1, 2]), List).

List = [A / [], B / [1, 2]]
A = A
B = B     More? (;)

List = [A / [1], B / [2]]
A = A
B = B     More? (;)

List = [A / [1, 2], B / []]
A = A
B = B
yes.
```

In fact, there are so many completely different uses for the attributed variables, that it is hard to give an exhaustive list of areas where they can be applied. Attributes or similar concepts have already been proposed in [10, 3, 7], but in **SEPIA** and **ECL^{i}PS^e** the metaterms are a primitive which is fully accessible to the user, independent of coroutining and fully supported by the system, which includes also the garbage collection and event handling.

2.5 Other Data Types

The metaterms are suitable for representing data that has a Prolog format and/or requires some particular control. Some extensions, however, may require data in a strictly non-Prolog format. An example can be numbers in double precision or IEEE format - which need several words of consecutive memory space. Most Prolog systems use tagged architecture where every word denotes a Prolog term and is tagged by a special combination of bits. Thus, to introduce a data type which occupies several consecutive words without tags, it would be necessary to define a new data type with a new tag, which would denote that the next n words in memory represent some particular object. Every time such a data type is introduced, the whole system kernel has to be

updated to take it into account, e.g. the compiler (for `assert/1`), the emulator (for unification), the garbage collector and built-in predicates.

To overcome this problem, we have defined in **SEPIA** the data type `string`, which represents a sequence of consecutive bytes in the memory. Although the system uses this data type only to represent sequences of ASCII characters, it in fact represents a buffer located on the heap (global stack) which can be used for any purpose. Its advantage is that it is already recognised by the whole system and only the output predicates give it a particular interpretation. So an extension writer can easily use it for various purposes.

Some built-in predicates raise an event if they encounter strings, in particular all arithmetic predicates. In this way it is straightforward to write packages that provide new number formats, e.g. infinite precision integers or rationals.

2.6 Modules

Modules are necessary for managing source programs as well as for providing a way to encapsulate run-time data into various compound objects. The former requires a static module system with a fixed structure (which can be easier compiled), but the latter needs a module system where new modules and their interfaces can be created, modified and erased at run time. The **SEPIA** module system matches the latter objective – it is predicate-based and it is fully dynamic, so that new items can be created and modified at any time. This mechanism can then be used to implement e.g. contexts.

3 Conclusion and Future Work

The architecture presented here has two main goals: simplicity and efficiency of extensions. We believe that the extensions must be written in a high level language so that they can be easily modified and further extended. It has often been our experience that the benefits from hard-coding a feature in C were short-lived, and the disadvantages of rigidity, complexity and decreased maintainability soon took over.

The lesson is that, while the system is being prototyped, even if it is a commercial prototype, as much as possible must be written in Prolog, and general solutions should be preferred to specialised ones. Only in the last step, when 'freezing' the system to a product, identified system bottlenecks can be hard-coded in C. As there is no such thing as 'a stable system' because all software is evolving, the high-level prototype can be further used for modifications and extensions of the product. All this may seem obvious, but a quick look at conference papers and common practice teaches us otherwise.

The architecture presented in this article describes the **SEPIA** system [9]. Recently, the **SEPIA** system has been merged with the MegaLog system [2] into **ECL^{i}PS^e** . **ECL^{i}PS^e** is fully compatible with **SEPIA** and keeps its extensibility, but it also has the MegaLog knowledge base, which makes its area of application wider and constitutes another type of support for extension writers.

Our experiences with the system are mostly positive. It was in fact quite surprising that it has taken some time to realize how extensible the system is and how its features should be correctly used. Sometimes we have written system parts in C and have later discovered that the available sockets already provided this functionality almost for free; with time we have learned to appreciate it. In our opinion, the fact that the **SEPIA** sources are not publicly distributed is not a drawback, because it imposes a discipline on extension writers to exploit the available functionality, rather than to modify the sources on every occasion. The benefits are obvious – apart from the ones mentioned at the beginning: an extension written completely in Prolog is more flexible and is compiled faster (the **SEPIA** compiler compiles about 2000 lines/sec.) than by recompiling sources or loading C code.

Since **ECL^{i}PS^e** is being used as the basis for most LP work at ECRC, we have received many suggestions and requests for further extensions. Some of the work planned for the near future includes:

- generalisation of the coroutining mechanism. Often users want to wake suspended goals in some particular order, usually the simple built-in predicates first, but sometimes a much more complex scheme is required. Another requirement is to give every woken goal a precedence and not to wake any goals with lower precedence while goals with higher precedences are executing.

  To accommodate such schemes, we have decided to shift much of the coroutining implementation to the Prolog level and leave the task of suspending and waking almost completely to the user. The machine will only raise an event when suspending a goal and when a suspending variable is bound.

  Although this change will almost certainly decrease the performance of some programs, users will be able to experiment with a much larger class of control strategies very easily. Successful strategies could then be coded again at a lower level (if necessary) to increase the performance again.

- Distributed coroutining. Coroutining can be quite naturally combined with interrupt processing. If an interrupt is processed synchronously, it can interact with the main execution and e.g. bind some variables and wake some goals. This will make it possible to extend coroutining to a distributed system where some goals are woken and executed on different machines and on success they bind the variables in the main process.

- Guard-like control. Although many applications require that suspending be performed at the procedure level, some applications or some predicates would benefit from a clause-level processing, i.e. suspending if the head unification or guard execution instantiates a variable.

- Since **ECL^{i}PS^e** will be used for various extensions in the CLP area, alternative number formats will be provided using the mechanism described above.

Bibliography

[1] Hassan Aït-Kaci and Roger Nasr. LOGIN: A logic programming language with built-in inheritance. *Journal of Logic Programming*, 3(3):185–215, 1986.

[2] J. Bocca. MegaLog – A Platform for Developing Knowledge Base Management Systems. In *Proc. Second Int. Symposium on Database Systems for Advanced Applications (DASFAA '91), Tokyo*, April 1991.

[3] Mats Carlsson. Freeze, indexing and other implementation issues in the WAM. In *Proceedings of the 4th ICLP*, pages 40–58, Melbourne, May 1987.

[4] Alain Colmerauer. Prolog II manuel de reference et modele theorique. Technical Report ERA CNRS 363, Groupe Intelligence Artificielle, Faculte des Sciences de Luminy, March 1982.

[5] M. Dincbas, P. Van Hentenryck, H. Simonis, A. Aggoun, T. Graf, and F. Berthier. The constraint logic programming language CHIP. In *International Conference on FGCS 1988*, Tokyo, November 1988.

[6] Pascal Van Hentenryck. *Constraint Satisfaction in Logic Programming.* MIT Press, 1989.

[7] Serge Le Huitouze. A new data structure for implementing extensions to prolog. In *PLILP*, pages 136–150, 1990.

[8] Micha Meier. Event handling in Prolog. In *Proceedings of the North American Conference on Logic Programming*, Cleveland, October 1989.

[9] Micha Meier, Abderrahmane Aggoun, David Chan, Pierre Dufresne, Reinhard Enders, Dominique Henry de Villeneuve, Alexander Herold, Philip Kay, Bruno Perez, Emmanuel van Rossum, and Joachim Schimpf. SEPIA - an extendible Prolog system. In *Proceedings of the 11th World Computer Congress IFIP'89*, San Francisco, August 1989.

[10] Ulrich Neumerkel. Extensible unification by metastructures. In *Proceedings of META'90*, 1990.

[11] James A. Thom and Justin Zobel. Nu-Prolog reference manual version 1.1. Technical Report 86/10, Department of Computer Science, University of Melbourne, 1986. Revised May 1987.