Implementing Prolog Programs Free from Unification

Paolo Frigo* Massimo Marchiori†

1 Introduction

In recent years many efforts have been spent in developing methods for efficiently execute Prolog programs. The usual approach to study improvements in the implementation of unification is based on *abstract interpretation* techniques. In this paper, we follow an alternative approach based on *syntactical analysis*, that is we restrict our attention to a subclass of Prolog programs, with the advantage of obtaining a very precise analysis of properties useful for compilation (like groundness, freeness and dereferencing).

The subclass we will implement is that of *Simply Well Moded* (SWM) programs. This class, introduced in [2], has the important property of being *unification free*, that is to say unification can be performed by iterated matching; moreover, as shown in the above paper, this class is quite large.

In order to implement efficiently SWM programs we start from the standard way of compiling Prolog, i.e. the Warren Abstract Machine (WAM), developed in [9] (however we will refer to the version presented by Aït-Kaci in his tutorial [1]). We show how, with some modifications both on the clause compilation scheme and on the instructions semantics, one can obtain an efficient and nice compiled code for these programs. In particular the following unpleasant WAM features disappear: the presence of free variables and reference chains, unsafe variables, the general `unify` operation, the `bind` operation, the trail structure and operations, and the `read/write` switch in all head instructions.

Since it is quite simple to test whether a program is SWM, we think an high performance compiler should be able to compile SWM programs according to our new scheme, otherwise it should apply other techniques in order to optimize the WAM code.

2 Simply Well Modedness

In this paper we will deal only with so-called *LD-derivations*, that is SLD-derivations in which the leftmost selection rule is used. Given a family S of objects (terms, atoms, etc.), $\mathrm{Var}(S)$ is the set of all the variables contained in it; moreover, S is said *linear* iff no variable occurs more than once in it. Given a substitution ϑ we indicate with $\mathrm{Dom}(\vartheta)$ its domain.

Definition 2.1 A substitution ϑ is said a *match* for two terms s and t if $(s = t\vartheta \wedge \mathrm{Dom}(\vartheta) \subseteq \mathrm{Var}(t)) \vee (t = s\vartheta \wedge \mathrm{Dom}(\vartheta) \subseteq \mathrm{Var}(s))$. We shall write $s \to t$ if there exists a match ϑ such that $s = t\vartheta$. □

In the usual treatment, a logic program is a finite set of clauses. Instead, we take as logic program a finite set of clauses together with a set of queries. This way we can talk about the derivations of a program P as the set of derivations given by the union of the derivations obtained from $P' \cup \{Q\}$, where P' is the greatest subset of P composed only by clauses, and Q ranges all over the queries in P. This formalism is well suited when formulating program properties, since it allows not to treat separately the clauses and the query.

Definition 2.2 A *mode* for a n-ary predicate p is a map from $\{1, \ldots, n\}$ to $\{+, -\}$. An argument position of a moded predicate is said of *input* (resp. *output*) if it is mapped by the mode into $+$ (resp. $-$). A *moded program* is a program with a map associating a mode to every predicate appearing in the program. □

¿From now on, we shall deal only with moded programs. Moreover we adopt the convention to write $p(\bar{s}, \bar{t})$ to denote a moded atom p having its input positions filled in by the sequence of terms $\bar{s}$, and its output positions filled in by $\bar{t}$. Now let us recall the following well-known properties:

Definition 2.3 A program is said *data definite* if every its computed answer substitution is grounding (see [7]). A moded program is said *data driven* if every time an atom is selected in a query during a derivation its input positions are filled in with ground terms. □

We start with the basic notion of well moding.

Definition 2.4 A moded program is said *Well Moded* (WM) if it satisfies the following conditions:

1. for every its query $?p_1(\bar{s}_1, \bar{t}_1), \ldots, p_n(\bar{s}_n, \bar{t}_n)$ it holds $\forall i \in [1, n] : \mathrm{Var}(\bar{s}_i) \subseteq \bigcup_{j=1}^{i-1} \mathrm{Var}(\bar{t}_j)$;
2. for every its clause $p_0(\bar{t}_0, \bar{s}_{n+1}) : -p_1(\bar{s}_1, \bar{t}_1), \ldots, p_n(\bar{s}_n, \bar{t}_n)$ it holds $\forall i \in [1, n+1] : \mathrm{Var}(\bar{s}_i) \subseteq \bigcup_{j=0}^{i-1} \mathrm{Var}(\bar{t}_j)$. □

*Dipartimento di Matematica, Università di Siena. Via del Capitano 15, 53100 Siena - Italy. `frigo@sivax.cineca.it`

†Dipartimento di Matematica Pura ed Applicata, Università di Padova. Via Belzoni 7, 35100 Padova - Italy. `max@hilbert.math.unipd.it`

Well Moded programs enjoy the following remarkable properties (see [3, 4]).

Theorem 2.5 *Well Moded programs are data definite and data driven.*

Now we introduce the class of programs that we will implement.

Definition 2.6 A moded program is said *Simply Well Moded* (SWM) if it is Well Moded and if it satisfies the following conditions:

1. for every its query $?p_1(\bar{s}_1, \bar{t}_1), \ldots, p_n(\bar{s}_n, \bar{t}_n)$ the family $\bar{t}_1, \ldots, \bar{t}_n$ is linear and composed only by variables;

2. for every its clause $p_0(\bar{t}_0, \bar{s}_{n+1}) : -p_1(\bar{s}_1, \bar{t}_1), \ldots, p_n(\bar{s}_n, \bar{t}_n)$ the family $\bar{t}_1, \ldots, \bar{t}_n$ is linear and composed only by variables and it holds $\mathrm{Var}(\bar{t}_0) \cap (\bigcup_{j=1}^{n} \mathrm{Var}(\bar{t}_j)) = \emptyset$. □

Example 2.7 The program for multiplying two natural numbers in Peano representation

$plus(0, N, N).$
$plus(s(L), M, s(N)) : -plus(L, M, N).$
$times(0, N, 0).$
$times(s(L), M, N) : -times(L, M, P), plus(P, M, N).$

is SWM by $times(+, +, -), plus(+, +, -)$. □

SWM programs have the property to be unification free, in the sense that the unification process during the resolution can be performed via repeated applications of the simpler pattern matching. More formally (we follow here [5]):

Definition 2.8 Two sequences of terms $s_1, \ldots, s_n$ and $t_1, \ldots, t_n$ with no variable in common are said *solvable by iterated matching* if, when they are unifiable, there is a permutation π of $\{1, \ldots, n\}$ and substitutions $\theta_1, \ldots, \theta_n$ such that for every k in $[1, n]$ θ_k is a match for $t_{\pi(k)}\theta_1 \cdots \theta_{k-1}$ and $s_{\pi(k)}\theta_1 \cdots \theta_{k-1}$.
A program is *unification free* if in every its derivation whenever a clause with head $q(\bar{s})$ is used to resolve the atom $q(\bar{t})$ selected in the query, then $\bar{s}$ and $\bar{t}$ are solvable by iterated matching. □

Theorem 2.9 *Simply Well Moded programs are unification free.*

The problem of unification freedom was initially studied in [6], extended by [2] and further studied in [5]. The interesting point is that this class is wide enough to include a large variety of programs (see the list included in [2]); moreover, in [5] it has been shown this class is optimal among the unification free classes that perform (iterated) matching in a deterministic form (in the sense that we know in advance the direction of the matching), an information that is readily important in optimization studies.

The intuition besides the SWM class is that all the output arguments of the body atoms are filled in with distinct 'fresh' variables, and hence unification can be performed with a 'double' matching, first on the input arguments (data driveness ensures we have a matching), and subsequently on the output arguments (having fresh variables, again, ensures matching is possible).

3 Clause Compilation

In the WAM the compilation scheme for a generic clause $p_0 : -p_1, \ldots, p_n$ is

```
unify arguments of p0;
write arguments of p1;
call p1;
   ⋮
write arguments of pn;
call pn
```

with the `allocate` and `deallocate` instructions, if needed. No hypothesis can be made on the unification in the head, so the instructions are very general (and expensive).

Our aim is to modify this scheme, and consequently the semantics of the instructions and the register allocation, in order to take advantage of the peculiar behavior of unification in SWM programs. On the contrary, the compilation of control (choice points, backtracking, indexing,...) will remain untouched, with the exception of the `trail` structure.

Let $p_0(\bar{t}_0, \bar{s}_{n+1}) : -p_1(\bar{s}_1, \bar{t}_1), \ldots, p_n(\bar{s}_n, \bar{t}_n)$ be a clause in a SWM program and suppose that it is used to solve a goal $?p_0(\bar{w}, \bar{z})$. As we have seen in Section 2 the unification can be replaced by the two matchings $\bar{t}_0 \leftarrow \bar{w}$, $\bar{s}_{n+1} \rightarrow \bar{z}$ because the terms in $\bar{w}$ are ground and $\bar{z}$ is made of fresh variables. Moreover, we know from Theorem 2.5 that the following resolution of $p_1, \ldots, p_n$ will make ground all terms in $\bar{s}_{n+1}$.

The standard WAM compilation of this kind of clauses presents two major sources of inefficiency. First of all, unification is used instead of matching. Secondly many variables are created in memory as unbound (i.e. autoreferencing cells), and then unified soon afterwards, before they can be aliased; thus, clearly, their initialization as unbound is unnecessary. For this reason these variables were called *uninitialized* by P. Van Roy in [8]. To avoid these inefficiences our solution is:

1. use matching to unify $\bar{t}_0$ and $\bar{w}$;

2. postpone the building (on the heap) of the terms $\bar{s}_{n+1}$, and the binding of $\bar{z}$ to them, until all the variables of $\bar{s}_{n+1}$ are instantiated to ground

terms, i.e. until the execution of the body goals $p_1, \ldots, p_n$ has been completed;

3. postpone the binding of the variables $\bar{t}_i$ to the outputs of the call to p_i after the call itself.

Hence the new compilation scheme is:

```
match t̄0 with argument registers X1,...,Xα0;
write s̄1 in argument registers X1,...,Xα1;
call p1;
copy Xα1+1,...,Xα1+β1 in the registers
in which the variables t̄1 are allocated;
  ⋮
write s̄n in argument registers X1,...,Xαn;
call pn;
copy Xαn+1,...,Xαn+βn in the registers
in which the variables t̄n are allocated;
write s̄n+1 in argument registers Xα0+1,...,Xα0+β0
```

where α_i is the number of input positions in p_i and β_i of the output ones (so $\alpha_i + \beta_i =$ arity of p_i).

It is of great importance to understand that the output variables $\bar{t}_i$ in a body goal $p_i(\bar{s}_i, \bar{t}_i)$ are not created on the heap as unbound before the call (as in the WAM); they are initialized only after the call, by coping $X_{\alpha_i+1}, \ldots, X_{\alpha_i+\beta_i}$ (that contain the ground terms to which the $\bar{t}_i$ are bound after the call) in the registers in which the $\bar{t}_i$ are allocated.

Now, having found a suitable clause compilation scheme for SWM programs, we shall study how to modify the WAM in order to let it follow this scheme.

4 Variables and Memory

The choice of the above scheme has a consequence of great importance for the WAM memory.

Claim 4.1 REF-*cells are no longer used, neither in the heap nor in the registers.*

We prove this claim by showing that in the new scheme all the WAM instructions that can produce REF-cells either are no longer used or, if used, do not actually produce REF-cells (in the first case we will drop them from the instruction set, in the second we will modify them by deleting their useless parts):

- `put_variable` and `set variable` could appear in the compilation of the body: however in the $\bar{s}_i$ $(i < n+1)$, by Definition 2.4, occur only variables already initialized (that are compiled with `put_value` and `set_value`), and the variables $\bar{t}_i$ $(i > 0)$ are initialized with ground values by the `copy` operations after the call to p_i;
- `set_void` is no longer used, because of Definition 2.6;
- `unify_variable` and `unify_void` could appear in the compilation of the head: however, for $\bar{t}_0$ we know by Theorem 2.5 that only the `read` mode is followed (while it is the `write` mode that produces REF-cells), and for $\bar{s}_{n+1}$, again by Theorem 2.5, all its variables have been already grounded in some previous part of the code;
- `unify_local_value`, `set_local_value` and `put_unsafe_value` can produce a new REF-cell only if a free REF-cell already exists on the stack: but this cannot be, as we have seen in the previous points.

The deep cause of Claim 4.1 is that in the procedural interpretation of a clause suggested in the new scheme a variable is grounded as soon as it appears. Thus, it shouldn't surprise that variables do not need to be represented in memory: only ground terms do!

Let us now list the effects of the absence of REF-cells on the whole WAM:

- reference chains disappear, so all `deref` operations can be removed;
- all `bind` operations can be removed;
- the `write` part in all `unify` and `set` instructions is no longer needed;
- the general `unify` operation can be reduced to a syntactical identity test between ground terms;
- all variables are *safe*[1], so all the instructions for the treatment of unsafe variables are no longer needed;
- the trail structure and all operations on it are no longer needed, because no variable is *conditional* (i.e. needs to be restored free in case of backtracking).

Variable classification and allocation rules have to be modified, according to the following facts:

- the code for $\bar{s}_{n+1}$ appears at the end of the clause code, so $\bar{s}_{n+1}$ has to be considered, as far as variables life is concerned, the last (i.e. rightmost) part of the clause;
- the call for a body goal p_i occurs between the code for $\bar{s}_i$ and $\bar{t}_i$;
- all variables are safe.

5 Instructions

In this section we shortly discuss which changes are needed in the WAM instructions.

[1] Recall that the existence of *unsafe* variables in the WAM is due to the interaction between autoreferencing stack cells and environment trimming.

5.1 Body Instructions

In the input arguments compilation the only difference from the standard WAM is that the instructions who allocate free variables in the memory (such as `put_variable`, `set_variable`,...) are no longer needed.

As regards the output arguments, after the execution of a call to a body goal $p_i(\bar{s}_i, \bar{t}_i)$ we expect that registers $\mathtt{X}_{\alpha_i+1}, \ldots, \mathtt{X}_{\alpha_i+\beta_i}$ contain the (ground) terms to which variables $\bar{t}_i$ must be bound. Thus we have only to copy their value in the registers $\mathtt{V}_1, \ldots, \mathtt{V}_{\beta_i}$ in which the $\bar{t}_i$ are allocated. Instead of introducing a new instruction `copy` $\mathtt{X}_j, \mathtt{V}_k \equiv \mathtt{V}_k \leftarrow \mathtt{X}_j$ we prefer to use the already existing `put_value` $\mathtt{X}_j, \mathtt{V}_k$, that has the same semantics. The code is then

```
call_p_i
put_value X_{α_i+1},V_1
    ⋮
put_value X_{α_i+β_i},V_{β_i}
```

5.2 Head Instructions

Now let us examine the most important modifications: those to head instructions.

Given a head $p_0(\bar{t}_0, \bar{s}_{n+1})$ of a clause in a SWM program, we know that input terms $\bar{t}_0$ have to be unified with ground terms represented in a dereferenced way in argument registers $\mathtt{X}_1, \ldots, \mathtt{X}_{\alpha_0}$. Then, as a general rule, we can compile $\bar{t}_0$ as in standard WAM, provided we modify the semantics by deleting the `deref` operation and the `write` part from all `get` and `unify` instructions. For example the new meaning of the instruction `get_structure` is

`get_structure` f/n,X_i $\equiv$
<tag,value>$\leftarrow \mathrm{X}_i$;
if (tag=STR) $\wedge$ (HEAP[value]=< f/n >)
  then
    begin
      S$\leftarrow$value+1;
      P$\leftarrow$P+*instruction_size*(P)
    end
  else *backtrack*

As regards the output, in the compilation scheme that we have adopted the code for the head output terms $\bar{s}_{n+1}$ is the last part of the whole clause code. Thus at the time it is executed all variables in $\bar{s}_{n+1}$ have been bound to ground terms (by Theorem 2.5). Moreover, the terms $\bar{s}_{n+1}$ have to be assigned to the registers $\mathtt{X}_{\alpha_0+1}, \ldots, \mathtt{X}_{\alpha_0+\beta_0}$ (and not to free heap cells, as happens in WAM). Then, instead of the `get` and `unify` instructions in `write` mode, we can use the `put` and `set` instructions, and compile $\bar{s}_{n+1}$ exactly as the body inputs $\bar{s}_i$.

We conclude this section with an example of compilation.

Example 5.1 $plus(s(L), M, s(N)) : -plus(L, M, N)$ with register allocation $L \rightarrow \mathtt{X}_1$, $M \rightarrow \mathtt{X}_2$, $N \rightarrow \mathtt{X}_4$ is compiled in

```
get_structure s/1,X_1
unify_value X_1
call plus/3
put_value X_3,X_4
put_structure s/1,X_3
set_value X_4
proceed
```

□

6 Conclusion

In this paper we have presented a new framework for compiling the class of Simply Well Moded programs. Using some theoretical results, we obtain very precise information on the computational behaviour of these programs, making useless the application of the usual techniques of abstract interpretation.

The essence of the framework can be summarized as follows: the unification of the input arguments is replaced with a matching while the unification of the output arguments is delayed until their final values are known. This way it is possible to get rid of some aspects of WAM that are now useless, allowing significative optimizations in the code.

Some hand-made calculations suggest that the speedups should range between 20% up to 50%, with respect to the standard WAM. However we are working to an implementation of the scheme which will allow us to sharply state its performances.

Another direction we are following consists in studying how to apply this scheme also to programs that are not SWM, by compiling separately their SWM parts. The main problem, to which we are presently working, is then the interleaving of the two kinds of code.

References

[1] H. Aït-Kaci. *Warren's Abstract Machine: a tutorial reconstruction*. The MIT Press, 1991.

[2] K.R. Apt and S. Etalle. On the unification free Prolog programs. In *Proc. MFCS'93*, LNCS, 1993.

[3] K.R. Apt and E. Marchiori. Reasoning about Prolog programs: from modes through types to assertions. In *Formal Aspects of Computing*, 1994. To appear.

[4] P. Dembinski and J. Małuszyński. AND-parallelism with intelligent backtracking for annotated logic programs. In *Proc. IEEE ISLP'85*, 1985.

[5] M. Marchiori. Localizations of unification freedom through matching directions. To appear in *Proc. ILPS'94*, 1994.

[6] J. Małuszyński and H.J. Komorowski. Unification-free execution of logic programs. In *Proc. IEEE ISLP'85*, 1985.

[7] U.S. Reddy. Transformation of logic programs into functional programs. In *Proc. IEEE ISLP'84*, 1984.

[8] P.L. Van Roy. *Can Logic Programming Execute as Fast as Imperative Programming?* PhD thesis, Berkeley, 1990.

[9] D.H.D. Warren. An abstract Prolog instruction set. Technical Note 309, SRI International, Menlo Park, CA, 1983.