An implementation of *dif* and *freeze* in the WAM

Mats Carlsson

SICS, Swedish Institute of Computer Science
PO Box 1263
S-16313 Spånga, Sweden

Phone: +46 - 8 - 7507970
E-mail: mats-c@sics.uucp

Abstract

Two very useful extensions to Prolog's computation model, *dif* and *freeze*, were introduced with Prolog II. A method for their incorporation into the Warren Abstract Machine is presented. Under reasonable assumptions, the method does not incur any overhead on programs not using these extensions.

Keywords: Logic programming, abstract machines, compilers, constraints, coroutines, Prolog.

§1 Introduction

We are considering the "New Engine", or WAM, for Prolog by D.H.D. Warren. In [Warren 83], an abstract machine is defined consisting of a memory model and about 40 instructions. We assume herein some familiarity with the WAM.

Prolog II [Colmerauer 82a,82b], has received much attention for its theoretical model as a term rewriting system in the domain of infinite trees, but also for its ability to delay certain predicates until enough information is available: *dif(X,Y)* , read as "*X* and *Y* are different terms", and *freeze(X,P)* , read as "delay *P* until *X* has been instantiated".

Similar primitives have subsequently been included into various implementations, for example LM-Prolog [Carlsson 83], MU-Prolog [Naish 85a], and NU-Prolog [Naish 86]. The concept of delay primitives is related to the idea of coroutines which were first introduced in Logic Programming with IC-Prolog [Clark 79], [Clark 80]. It is tightly coupled to the concept of constraints [Steele 80]. A study of the introduction of the constraint concept into Logic Programming is given in [Dincbas 86].

The usefulness of delay primitives is widely recognized. By adding a data-driven component to a language which otherwise is goal-driven, new programming techniques become available. They help solve certain semantical problems, in particular by allowing a sound treatment of negation. MU-Prolog, for example, supply sound versions built on delays for arithmetic, negation, and *if_then_else*. They can increase the efficiency of "generate and test" programs and even prevent infinite loops. They can be used for simulating and-parallelism by coroutining. Certain uses of delay primitives cause new yet unsolved semantical problems, for example when used for simulating perpetual processes computing with infinite streams. These problems are being attacked in a scheme called Constraint Logic Programming (CLP) [Jaffar 86], and some very promising results have already been reported.

The implementation issues have been addressed in [Naish 85b], in particular how to avoid complexity problems in implementations of *dif* . Naish also devises a method for automatic generation of wait declarations. Our work addresses the challenge of adding delay primitives to the WAM with reasonable efficiency without causing performance penalties for programs not using these primitives. We consider the work on constraint propagation techniques for logic programming [Dincbas 86] most significant, but it is outside the scope of this paper.

§2 Preliminaries

The terminology used herein will correspond to the procedural interpretation of logic programs, for example, relations are called *procedures* , body atoms are called *procedure calls* . We say that a

variable is *instantiated* when it is bound to a non-variable.

An important characteristic of the WAM is its use of *registers* for passing parameters and storing *temporary variables* , i.e. such that do not occur in more than one body goal. A variable that has next use after a body goal must be stored as a *permanent variable* in a binding environment, and it is the caller's responsibility to do so. As is customary, we denote temporary variables *1 .. n* by ***X1 .. Xn*** , which stand for global memory locations or machine registers. Permanent variables *1 .. m* are customarily denoted by ***Y1 .. Ym*** , which is short-hand for offsets from the start of the current environment.

The use of registers for parameter passing leaves considerable freedom for a compiler to optimize register usage. However, it causes some problems in implementing the ability to interrupt a computation and later resume from the interrupt. Such an ability is essential for *dif* and *freeze* . In general, interrupts may only occur when it can be determined which registers are live, for example at procedure calls, when the arity of the called procedure determines the set of live registers.

§3 Implementation Schemes

All implementations of delay primitives that the author is aware of are engineered in the same general framework: A new datatype, typically called *suspension* , is introduced. A suspension is an unbound variable with a reference to a suspended goal, represented as a record on the heap. A suspended goal is *woken* when the suspension is unified with another term.

Abstractly, we are partitioning the goal statement of SLD-resolution in two parts: (a) the ordinary goal statements, and (b) the suspended goals. The control structure of the inference engine is modified to deal with newly woken goals. The *termination condition* likewise has to be modified to preserve soundness: it no longer suffices that the ordinary goal statement is exhausted, since there could be goals that still are suspended. Not taking them into account would lead to incorrect answers. For example, in Prolog II, the programmer must ensure that goals are not suspended indefinitely.

3.1 Freeze in LM-Prolog

Implementations differ in details of the above framework. For instance, in LM-Prolog [Carlsson 83] suspended goals are woken recursively by the unifier, although it is implemented as a truth-valued function, without the ability to introduce choicepoints. This leads to an incomplete delay mechanism, since delayed goals are forcedly deterministic. In this paper we take a different approach where the unifier raises a condition which is tested at the next inference. Besides simplifying the interface between the engine and the unifier, this measure is necessitated by the precense of temporary WAM variables and the way that structures are allocated on the WAM heap.

Another detail where implementations differ is the unification of two suspensions. In LM-Prolog,

it is up to the unifier to combine the two suspensions into a single suspension over the conjunction of the two suspended goals. The approach taken in this work is to lay the responsibility on the suspension primitive and wake suspended goals whenever the suspended variable is *bound* rather than instantiated, whereupon the suspended goal may notice that it should re-suspend.

3.2 Dif in Prolog II

Superficially, Prolog II treats *dif* by a completely different mechanism. Prolog II replaces unification by the solving of systems of equations and inequations over infinite trees, using two algorithms called reduction and simplification [Colmerauer 84]. *dif(X,Y)* is defined as adding the inequation $X \neq Y$ to the system, and this has nothing to do with the *freeze* mechanism. However, the simplification algorithm, which is invoked whenever the system is augmented, can select certain inequations for "reprocessing", an operation that is tantamount to waking a delayed goal.

3.3 Boizumault's Framework

A proposed general framework for *dif* and *freeze* is given in [Boizumault 86]. The framework requires certain architectural extensions of the Prolog engine:

◊ an extra stack, the *frozen goal stack*, is added and contains pointers to frozen goals. The stack discipline makes deallocation at backtracking easy, but requires an extra slot in each choicepoint. Presumably, the stack discipline makes it easy to check termination.

◊ the *trail* entries are generalized to consist of <reference, previous value>. If this applies to all trail entries, it is a serious penalty on programs that do not use *freeze*. The generalization is motivated by examples like *:- freeze(X,p(X)), freeze(X,q(X))*. A suspension points to a *list* of frozen goals, and this list is destructively updated by the second *freeze*, and so the previous list has to be remembered.

In our approach, we avoid the extra memory area and choicepoint slot by simply using the heap. We trail each new suspension and check termination by scanning the trail. We avoid destructive updates and generalized trail entries by binding the old suspension to a new one which is a suspension over the conjunction *','(p(X),q(X))*.

§4 Freeze in the WAM

In addition to *dif* and *freeze*, we introduce a primitive *delay(X,P)* as the means for delaying the goal *P* until the variable *X* is bound to any other term *Y* even if *Y* is not instantiated. In *SICStus*, our experimental Prolog system developed at SICS, we have used *delay* as the basic delay mechanism and have built the other primitives on top of it. Van Caneghem also showed [Van Caneghem 84] that *dif* can be expressed in terms of other delay primitives.

4.1 The event mechanism

We introduce a general *event mechanism*, and implement *delay* as an instance of this mechanism. An event signals an extra-ordinary *condition* of some kind which needs to be dealt which at the

next inference. Typical conditions are interrupts from the terminal, trace mode in a debugger, the need to garbage collect, or the waking of a delayed goal. The reason for waiting until the next inference is that the event may involve a recursive computation. It is generally not possible for the engine to keep track of which argument registers are live, except when a predicate is entered, at which time the arity determines what registers are live. Thus the current sequence of WAM instructions is not interrupted until the next **call** or **execute** instruction.

A flag register ***F*** is introduced. ***F*** is initially zero and is cleared at backtracking. An event signals a condition by raising a bit in the flag register. When a predicate is entered, the flag register is tested:

```
Proc/n:        if F≠0
                 {
                   allocate;
                   Y1..Yn := X1..Xn;
                   Process F flags and reset F;
                   X1..Xn := Y1..Yn;
                   deallocate;
                 }
               <compiled code for Proc/n>
```

4.2 Wakeup events

Delay is conveniently implemented in the setting of events. The implementation requires a wakeup register *W* and a data structure *suspension over G*, where *G* is a goal. References to suspensions are called *constrained variables*. The ***W*** register contains the goal to wake at the next inference, if any.

A suspension is a heap data structure with two fields, denoted by `$susp(Valuecell,Goal)`. Constrained variables have their own datatype in the implementation. They are treated much like ordinary variables, i.e. are dereferenced by accessing the valuecell. However, when the valuecell is bound, a wakeup event is signalled, and the goal to wake is stored in ***W*** as:

```
               if F<wake>=0
                 {
                   W := Goal;
                   F<wake> := 1;
                 }
               else
                 W := ','(W,Goal);
```

That is, if more than one wakeup event is signalled before the next inference, a conjunction of goals to wake is built and stored in ***W***. The action taken at each inference when the wake flag is raised is simply to execute the contents of ***W***:

```
Proc/n:        :
```

```
:
if F<wake>=1
  {
    F<wake> := 0;
    X1 := W;
    call call/1,n;
  }
:
```

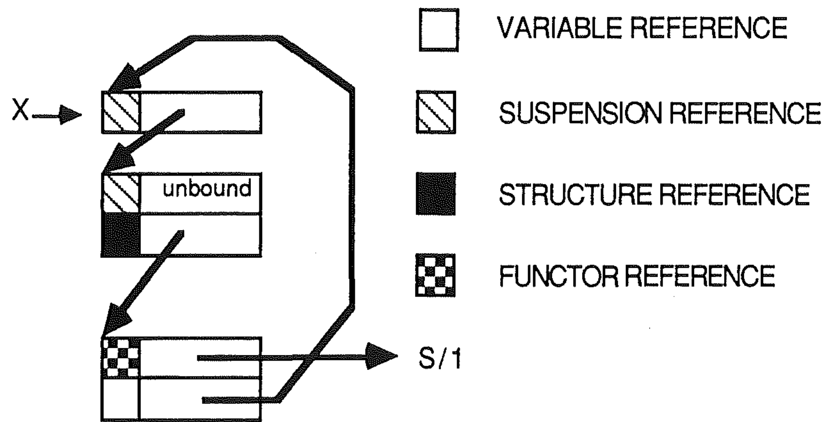
We can now give an operational definition of our primitive *delay(X,G)*, read as "delay *G* until *X* is bound", or "constrain *X* to satisfy *G* ". The definition deals with three cases: If *X* is instantiated, the goal *G* is invoked now. If *X* is a variable that is constrained to satisfy G_0, a suspension over the conjunction of G_0 and *G* is created. If *X* is an unconstrained variable, a suspension over *G* is created:

```
delay(X,G):
             if X is instantiated
               call(G);
             else if X=$susp(V,G0)
               bind V to $susp(_,',' (G0,G));
             else
               bind X to $susp(_,G);
```

To make the above account more explicit, we display here a few pictures of relevant data structures. Prolog terms are represented by tagged pointers. The first picture shows an ordinary WAM variable:

AN UNBOUND VARIABLE X

The second picture displays the situation when X has a constraint $s(X)$:

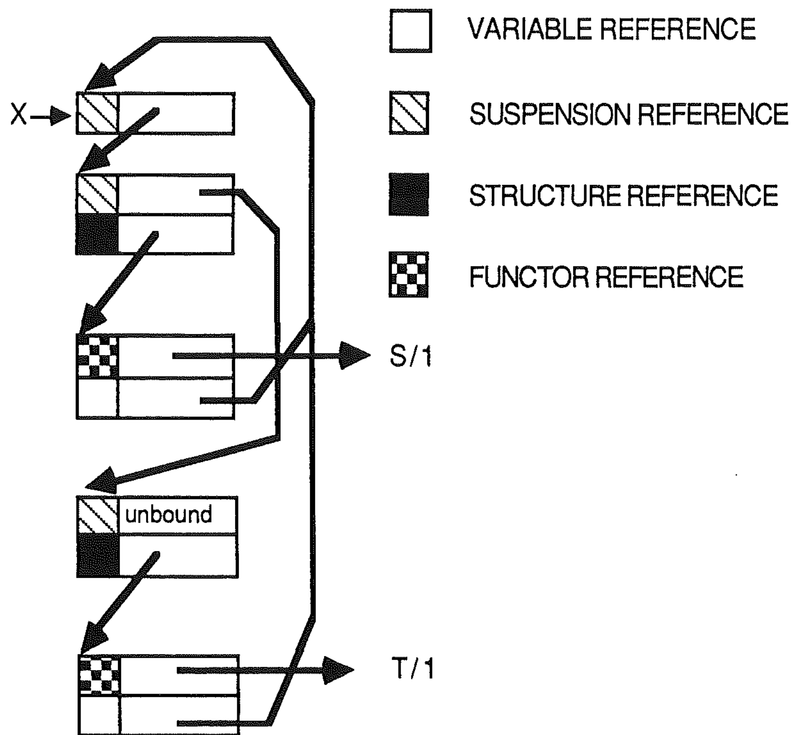


DATA STRUCTURES AFTER `delay(X,s(X))`

Thus the delay mechanism performs standard WAM actions: goals are copied to the heap, value cells are bound, and normal trail conditions apply. The only extra work needed at backtracking is to clear the ***F*** register.

Note that in this implementation, as opposed to LM-Prolog and [Boizumault 86], the case where two constrained variables are unified is not treated as a special case. Instead, it is the responsibility of *delay* to deal with that case.

Finally, if *X* had two constraint *s(X)* and *t(X)*, the corresponding data structures would be as in the picture:



DATA STRUCTURES AFTER `delay(X,s(X)), delay(X,t(X))`

4.3 Detection of Termination

We use the *trail* for keeping a record of all suspensions that have been created simply by pushing a reference to each new suspension on the trail. The toplevel can then detect unsatisfied delayed goals by scanning the trail.

Trail entries are touched by the engine at two other occasions:

◊ When the engine backtracks, all references in a section of the trail are reset to unbound. The extra suspension references cause no problem here, since unbound suspensions are (vacuously) being reset to unbound.

◊ At the cut operation, such references in a section of the trail that refer to a certain part of the heap and stack are deleted. Because of the termination detection, this algorithm has to be modified to *not* delete references to unbound suspensions. We argue that this should not cause any extra overhead for the cut operation, if suspensions are represented as a unique basic type.

The termination test could be encapsulated in a generalized metacall primitive *implies(S,G)* which tries to solve G :

- if G fails, S = `false`
- if G succeeds and no delayed goals exist, S = `true`
- otherwise, S = < a conjunction of unsatisfied delayed goals >

4.4 Inline Goals

Since raising a wake condition and waking a goal happen at different times, there is a problem with the instructions executed before the next inference, if those instructions can cause failure or remove choicepoints, e.g. arithmetic tests, meta-logical type tests, and cut. This problem occurs for *inline goals* that compile to instructions rather than procedure calls. For example, in

```
p(1,X) :- !, q(X).
p(2,2).

:- dif(U,1), p(U,X).
```

if the cut compiles inline and is executed before waking `dif(U,1)`, the solution (`X = U = 2`) is not found. In

```
p(1,X) :- var(X).
p(2,X) :- nonvar(X).

:- freeze(U,U=V), p(U,V).
```

if *var* compiles inline the incorrect solution `U = 1` is found. We avoid this problem by *decompiling* [Buettner 86] inline goals, so as to delay them if a wake condition has been raised, thus preserving the desirable procedural semantics of delayed goals. For example, cut compiles to an instruction `cut(X)` with semantics

```
cut(X):      if F<wake>=1
               W  := ','(W,cut(X));
             else
               <perform normal cut operation>
```

and similarly for all instructions corresponding to inline goals. This ensures that the cut is properly delayed until after the goal W. The emulator overhead of testing the wake condition can be avoided by well-known techniques.

We thus use decompilation for constructing a goal which is <u>semantically</u> equivalent to an instruction sequence, although not necessarily <u>syntactically</u> identical with the source code.

4.5 Conjunctive and Disjunctive Freeze

Several applications require the ability to delay until *each* element, or *some* elements, of a set of variables have been instantiated. For instance, a dataflow instruction waits until all arguments have arrived. An equation solver waits until enough unknowns have been instantiated.

Delaying *p(X,Y)* until *X* and *Y* have both been instantiated is easily expressed by:

```
:- freeze(X,freeze(Y,p(X,Y))).
```

Delaying *p(X,Y)* until either *X* or *Y* have been instantiated can be expressed by (this example is adapted from the Prolog II manual):

```
:- freeze(X,thaw(M,1,p(X,Y))),
   freeze(Y,thaw(M,2,p(X,Y))).


thaw(M,M,G)  :- call(G).
thaw(M,N,_)  :- M ≠ N.
```

i.e. using *M* as a mutual exclusion variable, ensuring that *p(X,Y)* is not woken twice. If this second technique is used, the termination test has to be trivially modified to avoid spurious warnings about goals of the form:

```
freeze(_,thaw(_,_,_)).
```

Similar but more hard-wired mutual exclusion techniques are used in the implementation of FCP [Mierowsky 85].

4.6 Freeze and Dif in terms of Delay

It should be clear how to define *freeze* and *dif* in terms of *delay*. *Freeze* could be defined as:

```
freeze(X,G) :- var(X), delay(X,freeze(X,G)).
freeze(X,G) :- nonvar(X), call(G).
```

Dif could be defined as:

```
dif(X,Y) :-
  different(X,Y,M),
  dif'(M,X,Y).

dif'(true,_,_).
dif'(maybe(M),X,Y) :- delay(M,dif(X,Y)).
```

where *different(X,Y,M)* is true if:

M = `true` and *X* and *Y* do not unify, or
M = `false` and *X* and *Y* are identical, or
M = `maybe(V)` and the variable *V* has to be bound for *X* and *Y* to unify.

Both of these definitions have the property that binding the constrained variable causes the suspension to wake up, the test to be re-computed, and possibly a new suspension to be created. This potential inefficiency has been discussed in [Naish 85b], where remedies to the problem are proposed.

One could argue for a more basic *freeze(X,P)* mechanism that waits until X is instantiated to avoid the recomputation of *var(X)*. Such a mechanism does however not suffice for *dif*. For example, in

```
:- dif(X,Y), X=Y.
```

the failure would go undetected, i.e. *dif* will stay suspended, since *X* and *Y* are uninstantiated. LM-Prolog uses *freeze* as the basic mechanism with a special case treatment of *dif* to avoid this bug.

4.7 Language issues

We agree with [Naish 85b] that explicit calls to *freeze* tend to obscure the logic of programs since a call *freeze(X,P)* is logically equivalent to *P* . We therefore use a variant of the !-annotations of IC-Prolog. A declaration

```
:- mode Pred(!,?,!).
```

denotes that any call *Pred(X,Y,Z)* with *X* or *Z* uninstantiated must delay. In IC-Prolog, the annotation occurs on goal variables rather than in a declaration. We chose to use a declaration instead, to simplify compilation. Note that our declarations are much less sophisticated than MU-Prolog's in that subarguments cannot be declared, and the delay criterion is tested before any head unification is attempted. The above declaration is treated as an abbreviation for extra clauses added to `Pred/3` as follows:

```
Pred(A,B,C) :- var(A), !, delay(A,Pred(A,B,C)).
Pred(A,B,C) :- var(C), !, delay(C,Pred(A,B,C)).
```

In the implementation, wait declarations are compiled more compactly using special wait instructions which precede the indexing code:

```
Pred/3:     wait(Pred/3,X3)
            switch_on_term Lv,Lc,Ll,Ls
Lv:         wait_x1(Pred/3)
Lc:         /* constant case */
Ll:         /* list case */
Ls:         /* structure case */
```

where the semantics of the new instructions are:

`wait(F/n,Xi)` : If *Xi* is uninstantiated, then construct on the heap a goal *G = F(X1,.. Xn)* and execute *delay(Xi,G)*. If *Xi* is instantiated, then continue at the next instruction.

`wait_x1(F/n)` : X1 is guaranteed to be uninstantiated. Construct on the heap a goal *G = F(X1,.. Xn)* and execute *delay(X1,G)*.

§5 Concluding Remarks

We have compared some implementations of delay primitives like *dif* and *freeze* and proposed a way of incorporating them in the WAM. The proposal is built on a more general event mechanism which can be argued to be necessary for handling conditions like user interrupts and garbage collections. Assuming the presence of an event mechanism and the availability of a new datatype, the WAM extension does not cause any execution time penalty for programs that do not use the extensions. By combining the event mechanism with decompilation techniques, the correct execution sequence for delayed and inline goals is preserved.

Acknowledgements

Many of the ideas presented herein are part of "Prolog folklore" and are difficult to attribute to specific individuals. Intellectually, a lot is owed to the pioneer work of the Groupe Intelligence Artificielle at Marseille. The author is also indebted to Jonas Barklund, Ken Kahn, Lee Naish, Ross Overbeek, Jeff Schultz, and to his colleagues at SICS for discussing and refining the ideas reported herein. The author is aware that NU-Prolog contains something very similar to the implementation reported herein but he does not know any details.

References

[Boizumault 86] P. Boizumault, *A general model to implement DIF and FREEZE,* Proc. 3ICLP, London, 1986.

[Buettner 86] K.A. Buettner, *Fast Decompilation of Compiled Prolog Clauses* , Proc. 3ICLP, London, 1986.

[Carlsson 83] M. Carlsson, K.M. Kahn, *LM-Prolog User Manual* , UPMAIL Technical Report #24, Dept. of Computing Science, Uppsala University, 1983.

[Clark 79] K.L. Clark, F.G. McCabe, *The Control Facilities of IC-Prolog* , in {D. Michie, *Expert Systems in the Microelectronic Age* , University of Edinburgh, 1979}.

[Clark 80] K.L. Clark, F.G. McCabe, *IC-Prolog - Language Features* , in {K.L. Clark, S-Å. Tärnlund (eds), *Logic Programming,* Academic Press, 1982}.

[Colmerauer 82a] A. Colmerauer, *Prolog and Infinite Trees,* in {K.L. Clark, S-Å. Tärnlund (eds), *Logic Programming,* Academic Press, 1982}.

[Colmerauer 82b] A. Colmerauer, *Prolog II: Manuel de Référence et Modèle Théorique,* Groupe Intelligence Artificielle, Université Aix-Marseille II, 1982.

[Colmerauer 84] A. Colmerauer, *Equations and Inequations on Finite and Infinite Trees,* Proc. International Conference on FGCS, Tokyo, 1984.

[Dincbas 86] M. Dincbas, *Constraints, Logic Programming and Deductive Databases* , Proc. France-Japan Artificial Intelligence and Computer Science Symposium, Tokyo, 1986.

[Jaffar 86] J. Jaffar, J-L. Lassez, M.J. Maher, *Some Issues and Trends in the Semantics of Logic Programming* , Proc. 3ICLP, London, 1986.

[Mierowsky 85] C. Mierowsky, S. Taylor, E. Shapiro, J. Levy, M. Safra, *The Design and Implementation of Flat Concurrent Prolog* , Weizmann Institute Technical Report CS85-09, 1985.

[Naish 85a] L. Naish, *The MU-Prolog 3.2 Reference Manual* , Dept. of Computer Science, University of Melbourne, 1985.

[Naish 85b] L. Naish, *Negation and Control in Prolog* , Ph.D. Thesis, Dept. of Computer Science, University of Melbourne, 1985.

[Naish 86] L. Naish, *Negation and Quantifiers in NU-Prolog* , Proc. 3ICLP, London, 1986.

[Steele 86] L.G. Steele, *The Definition and Implementation of a Computer Programming Language based on Constraints*, Ph.D. Thesis, MIT, 1980.

[van Caneghem 84] M. van Caneghem, *L'anatomie de Prolog-II* , Thèse, Université Aix-Marseille, 1984.

[Warren 83] D.H.D. Warren, *An Abstract Prolog Instruction Set*, SRI International #309, 1983.