

Suspension Frames on the WAM Heap

Bart Demoen, Phuong-Lan Nguyen

Report CW 471, December 2006



Katholieke Universiteit Leuven
Department of Computer Science
Celestijnenlaan 200A – B-3001 Heverlee (Belgium)

Suspension Frames on the WAM Heap

Bart Demoen, Phuong-Lan Nguyen

Report CW471, December 2006

Department of Computer Science, K.U.Leuven

Abstract

A suspension encapsulates everything needed to execute a piece of code many times in the same context: the need for this feature exists in constraint solver programming, where (part of) a predicate checking the satisfiability of a set of constraints needs to be re-executed every time the domain of a concerned variable changes. Traditional extensions of the WAM support a heap term and meta-call based approach to this issue. The most efficient implementation of suspensions in the context of Prolog exists in B-Prolog. B-Prolog is based on the TOAM and suspension frames are put in the execution stack. They are notoriously cheap to re-enter. It is clear that the WAM could be adapted to allow suspension frames on the local stack. However, it is worth exploring a scheme that puts suspension frames on the WAM global stack (the heap). We describe in detail the few additions to a traditional WAM system to achieve this. Together with attributed variables that is enough to compile B-Prolog's Action Rules to almost plain Prolog. We have enhanced hProlog with the required features to support suspension frames on the heap. We compare the performance of our Action Rules implementation with B-Prolog on a set of benchmarks previously used by the author of Action Rules.

Suspension Frames on the WAM Heap

Bart Demoen* and Phuong-Lan Nguyen†

* Department of Computer Science, K.U.Leuven, Belgium
bmd@cs.kuleuven.be

† Institut de Mathématiques Appliquées, UCO, Angers, France
nguyen@ima.uco.fr

Abstract. A suspension encapsulates everything needed to execute a piece of code many times in the same context: the need for this feature exists in constraint solver programming, where (part of) a predicate checking the satisfiability of a set of constraints needs to be re-executed every time the domain of a concerned variable changes. Traditional extensions of the WAM support a heap term and meta-call based approach to this issue. The most efficient implementation of suspensions in the context of Prolog exists in B-Prolog. B-Prolog is based on the TOAM and suspension frames are put in the execution stack. They are notoriously cheap to re-enter. It is clear that the WAM could be adapted to allow suspension frames on the local stack. However, it is worth exploring a scheme that puts suspension frames on the WAM global stack (the heap). We describe in detail the few additions to a traditional WAM system to achieve this. Together with attributed variables that is enough to compile B-Prolog's Action Rules to almost plain Prolog. We have enhanced hProlog with the required features to support suspension frames on the heap. We compare the performance of our Action Rules implementation with B-Prolog on a set of benchmarks previously used by the author of Action Rules.

1 Introduction

This report is meant to contain mostly technical details and explanations of what it takes in a WAM context to keep suspension frames on the heap. Explanations of the WAM, TOAM, Action Rules, attributed variables ... are not reproduced here. See for instance [7, 1, 8, 11, 5, 2].

The aim of this paper is three-fold:

- show the modifications needed to implement suspension frames on the WAM heap; this approach was implemented in hProlog
- show how the primitives used for suspensions on the heap can be used to implement Action Rules
- compare experimentally our implementation of Action Rules with the one in B-Prolog; for this we use a small set of benchmarks formerly used in [9] and a set of artificial benchmarks measuring particular aspects of suspension frames

We use hProlog 2.7, a descendant of dProlog [3], which we have used before for experiments. The TOAM is only implemented in B-Prolog: we have used Version 6.9 #1. We used SICStus Prolog 3.12.0 as well.

The paper is structured as follows: Section 2 explains the basic implementation schema for keeping suspension frames on the WAM heap: we are not concerned at that point with higher level constructs like `freeze/2` or Action Rules. That section only describes a basic mechanism for setting up a suspension associated to a goal, so that re-entering the clause(s) of the corresponding predicate is cheap. Section 3 shows how Action Rules can be compiled to the WAM with attributes and suspension frames on the heap. The nice thing here is that we can describe this as a source to source transformation. Section 4 compares the efficiency of WAM suspension frames in hProlog with B-Prolog. Section 5 points out the fingerprint and fills in some gaps in the basic schema. In particular we discuss how choices related to determinacy of the Action Rules bodies and recursive activation of suspended frames affect the implementation. Section 6 briefly discusses some alternatives. We conclude with Section 7.

2 Suspensions on the WAM heap: three built-ins and a declaration

We start with a very simple example of a single-clause predicate and an accompanying query:

```
:- suspension(foo/1).

foo(Susp) :-
    re_entry_point(Susp),
    writeln(hello),
    go_back.

?- foo(Susp), re_entry(Susp), re_entry(Susp), re_entry(Susp).
```

The idea is that

- the call `foo(Susp)` allocates an environment: the suspension declaration results in this environment being allocated on the heap; this environment is named a *suspension frame*
- `re_entry_point/1` unifies its argument with a datastructure holding enough information to restart execution in the environment of the call to `foo(Susp)` and just after the goal `re_entry_point(Susp)`; goal `re_entry_point(Susp)` also transfers control immediately to the caller of `foo(Susp)` *without* deallocating the environment; the datastructure is named a *suspension term*
- an invocation of `re_entry(Susp)` sets the current environment pointer `E` to the environment of the `foo(Susp)` call, and transfers control to the sequence of instructions for the `writeln(hello)` goal

- the call to `go_back` transfers control to the caller of `re_entry(Susp)` *without* deallocating the environment

The result of the query is that *hello* is written three times.

This example doesn't show the benefits of the suspension on the heap, but it shows the essentials of the mechanism: the environment stays on the heap and can be re-used as many times as one likes.

The compiler (Prolog source code to WAM code) needs very little change: the `allocate` instruction for `foo/1` is replaced by a new instruction `alloc_heap`. The emulator just needs three new built-ins: `re_entry_point/1`, `re_entry/1` and `go_back/0`. Note that we do not intend to make these new built-ins available to the user as their undisciplined use causes havoc: they are meant to make a source to source transformation feasible, e.g. from Action Rules to Prolog with a few extensions.

3 Transforming Action Rules to Prolog: an example

In [5] we have attempted to precisise the semantics of Action Rules by a program transformation to almost plain Prolog¹. The schema presented there was based on a traditional meta-call based approach to delay. We can adapt that transformation schema to one that uses suspension frames. However, [5] also caters for the `event/2` trigger, and this blurs the picture somewhat. So we prefer to present here directly the transformation from Action Rules as far as they are used for specifying `freeze/2` to code in hProlog that uses the new primitives.

We start from the following Action Rules example program:

```
foo(X,_,_), var(X), {ins(X)} => true.
foo(X,A,Z) => a(A,B), b(B,Z).
```

The goal `foo(X,A,Z)` behaves like the goal `freeze(X,(a(A,B), b(B,Z)))`.

Here is the translation of the above Action Rules into Prolog code which calls the new built-in predicates:

```
:- suspension(foo/3).

foo(X,A,Z) :-
    (var(X) ->
        attach_suspension(X,Susp),
        re_entry_point(Susp),
        a(A,B),b(B,Z),          % (*)
        go_back
    ;
        a(A,B),b(B,Z),
        go_back
    ).
```

¹ This is Prolog with attributed variables.

First note that we reserve the attribute slot in an attributed variable for triggering on instantiation only: it will be clear how to deal with richer attributed variables later.

We go over the new ingredients once more, providing more detail:

- The declaration `:- suspension(foo/3)`. indicates to the compiler that the code for the (single) clause for `foo/3` must start with the instruction `alloc_heap` which works like the WAM `allocate` instruction, but which allocates the environment on the heap.
- The goal `attach_suspension(X,Susp)` attaches `Susp` (at this point still a free variable) to the attribute of `X` - if `X` was already an attributed variable with suspensions attached to it, then `Susp` is placed in conjunction with those suspensions: we will come back to its exact implementation later. In general, `attach_suspension(X,Susp)` must attach a suspension to every variable in the term `X`.
- The goal `re_entry_point(Susp)` performs two actions:
 1. `Susp` is unified with a datastructure (the suspension term) that holds the current environment pointer (i.e. a pointer to the frame just allocated on the heap) and a code pointer that points just after the call to `re_entry_point(Susp)`. The suspension term may contain further information, e.g. for deactivating (or killing) the suspension.
 2. control is transferred back to the caller of `foo/3` without deallocating the frame
- The goal `go_back` also transfers the control back to the caller of `foo/3` without deallocating the frame

At some time after the goal `foo(X,P,Q)` was executed, `X` is instantiated and the conjunction `a(A,B),b(B,Z)` (on the line with `(*)`) is activated. This happens as follows: when `X` is instantiated, the unify hook for `X` is called; it gets the suspension term from `X`; it sets the WAM E-pointer to the frame pointer from the suspension term, and the program counter to the code pointer from this suspension term. The new built-in predicate `re_entry(Susp)` performs these actions. The unify hook eventually calls:

```
deal_with_wakeup(Susp,X,Cont) :-
    (attvar(X) ->
        [...] % attach conjunction to X
    ;
        re_entry(Susp)
    ),
    call(Cont).
```

Finally, we take a look at the implementation of `attach_suspension/2` and how to construct a conjunction of suspensions. Note that `attach_suspension/2` is called only with a free first argument. The first argument can be an attributed variable, in which case it already carries a suspension.

```

attach_suspension(X,NewSusp) :-
    (attvar(X) ->
        get_attr_lowlevel(X,Susp),
        conj(Susp,SuspNew,Conj),
        put_attr_lowlevel(X,Conj)
    );
    make_new_attvar(X),
    put_attr_lowlevel(X,NewSusp)
).

:- suspension(conj/3).

conj(Susp1,Susp2,Conj) :-
    re_entry_point(Conj),
    re_entry(Susp1),
    re_entry(Susp2),
    go_back.

```

All ingredients have been explained before, so we only give a high level explanation: a conjunction of suspension terms is just a new suspension term, which corresponds to a frame on the heap for the conj/3 clause. There is a more traditional alternative to this construction: the term (Susp1,Susp2) would also be useful as a conjunction. However, with the current definition, the conj/3 predicate blends in perfectly with the rest of the suspension on the heap approach. There is a performance penalty when the conjunction is never executed of course.

Nested freeze/2 calls Often code contains nested calls to freeze/2. Here is a goal from the sort benchmark:

```
freeze(X,freeze(Y,X=<Y)),
```

In [6] we have explained that the naive execution of such a nested freeze goal is very inefficient. Here is a form that is more efficient and which we named *symmetric* for obvious reasons: replace the above goal by

```
Goal = smaller(X,Y,Z), freeze(X, Goal), freeze(Y, Goal),
```

and define

```

smaller(X,Y,Z) :-
    (nonvar(X), nonvar(Y), var(Z) ->
        X =< Y,
        Z = 1
    );
    true
).

```

In the benchmarks, we use the symmetric form in the benchmarks `sort`, `send` and `queens`: `nrev` does not contain nested freeze calls.

There are (at least) two ways to transform such nested freeze calls to suspension frame code. Here are the two forms we singled out for suspension frame code - we use the same example:

The first form is verbose and attempts to minimize the amount of testing after a wakeup:

```
:- suspension(smaller/2). % minimize testing after wakeup

smaller(X,Y) :-
    (var(X) ->
        (var(Y) ->
            attach_suspension(X,SXY), attach_suspension(Y,SXY),
            re_entry_point(SXY),
            (nonvar(X), nonvar(Y) ->
                X =< Y, kill_go_back(SXY)
            );
            go_back
        )
    );
    attach_suspension(X,SX),
    re_entry_point(SX),
    X =< Y, go_back
);
    (var(Y) ->
        attach_suspension(Y,SY),
        re_entry_point(SY),
        X =< Y, go_back
    );
    X =< Y, go_back
).
```

For the moment, read `kill_go_back(SXY)` simply as `go_back`. This point is covered in detail in Section 5.

The second form corresponds to the symmetric freeze form:

```

:- suspension(smaller/2).    % symmetric form

smaller(X,Y) :-
    (var(X) -> attach_suspension(X,SXY), P = 1 ; true),
    (var(Y) -> attach_suspension(Y,SXY), P = 1 ; true),
    (var(P) ->
        X =< Y,
        go_back
    ;
        re_entry_point(SXY),
        (nonvar(X), nonvar(Y) ->
            X =< Y,
            kill_go_back(SXY)
        ;
            go_back
        )
    ).

```

The size of the latter form is linear in the number of variables involved², while the former form is exponential in the number of variables. B-Prolog has chosen a different trade-off: it generates code that is quadratic in the number of involved variables.

For the real benchmarks, the two suspension forms perform largely the same: most often all the variables are free at the moment of the call.

4 Suspension frames on the TOAM stack versus the WAM heap

4.1 *Real* benchmarks

We use the same set of benchmarks as in [9]: nrev³, send⁴, sort⁵ and queens⁶. The B-Prolog distribution contains these programs in a version with freeze/2 and a version with delay clauses. We have adapted the version with delay clauses so as to use Action Rules. When appropriate, we use the *symmetric* version of the same benchmark: [6] goes into some detail to explain why that is reasonable. The Action Rule version is also transformed into a version using the suspension on the heap. The relevant parts of the benchmarks are reproduced in the Appendix.

² The sendmoremoney benchmark contains a nested freeze call with 5 levels, so it is worth paying attention to this.

³ Repeated reversing a list of 500 elements

⁴ The send-more-money puzzle.

⁵ Sorting a list of 19 elements.

⁶ Finding all solutions to the 11-queens problem.

	B-Prolog freeze	B-Prolog Action Rules	hProlog symmetric	hProlog suspension	SICStus symmetric	SICStus block
nrev	456	452	268	328	1590	750
send	294	322	236	216	1090	500
sort	1468	1476	1060	956	5000	2110
queens	1298	1196	936	1086	5410	2840

Table 1. Comparing suspension frames on the heap with Action Rules and freeze/2

We show the figures for SICStus Prolog as well, so that the reader can see that the performance of B-Prolog and hProlog is not too bad. We We have also measured Yap 5.1.1: it was almost always significantly slower than SICStus.

hProlog is always noticeably faster than B-Prolog: this is also true for programs not using delay. The figures show that the TOAM does not give a systematic edge for programs using delay over a WAM implementation. The suspension frame on heap approach seems indeed competitive with the TOAM approach with suspensions on the local stack. Within hProlog, the performance figures do not give a clear guidance which to prefer: the pure term approach or the suspension frame approach. However, it shows that our approach to conjunctions of suspensions is not to be snared at.

Nrev is a bad case for suspension frames on the WAM heap: each suspension is activated only once and the performance is dominated by the creation of the suspension frames and suspension terms. Making creation faster must therefore be a priority in future work.

4.2 Artificial benchmarks

We tried to measure separately the performance of freezing a goal on a variable, freezing a sequence of goals on the same variable (resulting in a conjunction of frozen goals on one variable), melting one goal attached to a variable and melting a conjunction of goals on a variable. The previous sentence contains classical WAM freeze style terminology. The corresponding Action Rule or suspension actions should be clear.

More precisely, table 2 shows performance figures for

- *one freeze/melt**: freeze(X, \bar{g}), repeat($N*N$), $X = 1$
- *freeze* (no conj)*: repeat($N*N$), freeze(X, \bar{g})
- *freeze* (conj)*: repeat(N), do_n_times(N , freeze(X, \bar{g}))
- *conj freeze*/melt**: do_n_times(N , freeze(X, \bar{g})), repeat(N), $X = 1$

with value $N = 1000$, and \bar{g} equal to a goal with two arguments.

The figures for SICStus Prolog are there just for giving more perspective, but we are not commenting on them.

The figures for B-Prolog freeze can be left out of the discussion: they perform quite close to action rules, and B-Prolog’s thing is really action rules.

Table 2 shows that

	B-Prolog Action Rules	B-Prolog freeze/2	hProlog freeze/2	hProlog suspensions	SICStus freeze
one freeze/melt*	630	712	481	394	1616
freeze* (no conj)	378	334	240	311	436
freeze* (conj)	425	373	129	436	340
conj freeze*/melt*	302	365	184	188	990

Table 2. Comparing suspensions with Action Rules and freeze/2 on artificial benchmarks

- hProlog’s suspensions perform similar or better than B-Prolog’s action rules; only in forming conjunctions of agents waiting for the same variable to be instantiated does B-Prolog perform slightly better; melting a goal seems significantly faster for hProlog’s suspensions
- in hProlog, suspensions perform better than freeze for melting single goals and only slightly worse when melting conjunctions; however, the freezing action itself is more expensive; this would tip the scale to suspensions in cases that re-activation of the agent dominates

Caveat The interpretation of the benchmarks is not complete without knowing that in hProlog we have used the single attribute slot for the freeze attribute or the suspension term: other attributes cannot be used at the same time at this moment. We intend to remove that restriction and - based on preliminary experiments - we expect only a marginal slowdown.

5 Some fingerprint of suspension frames on the WAM heap

Several issues have not been covered yet: we deal with recursive activation of suspensions in Section 5.3, the determinism of an action rule body (Section 5.2), garbage collection (Section 5.4), killing a suspension (Section 5.1), corouting (Section 5.5), and the management of the top of the local stack (Section 5.6).

5.1 Killing an agent

In B-Prolog an agent is killed when the conditions of an action rule succeed and this action rule has no events, i.e. the action rule is just a matching clause. In the previous examples this happens when the last action rule is selected. Killing prevents the same agent to be executed again and is necessary so that agents can be garbage collected and to prevent execution of agents that would not further contribute to solving of the problem. In the suspension frame approach in hProlog, this killing happens by the built-in *kill_go_back/1*: its argument is a suspension term. The suspension term contains a slot to indicate whether the suspension is still alive. *Kill_go_back/1* sets it to dead and a call to *re_entry/1* first checks this slot. Sometimes it is not necessary to kill a suspension term: if only one variable has the suspension term, and the trigger is instantiation,

the suspension can only be activated once. This is the case in situations like *freeze(X,Goal)*. The marking phase of garbage collection just doesn't find any references to the suspension term and it can be collected together with the suspension frame itself. When a goal is frozen on two or more variables - as in the symmetric version of the nested *freeze(X,freeze(Y,Goal))* - killing is necessary because one unification (e.g. $f(X,Y) = f(1,2)$) can activate the same suspension twice.

5.2 Determinism

In B-Prolog, the body of an action rule is not allowed to leave choicepoints. This is enforced by the implementation: a cut is placed at the end of the body. On one hand, this makes implementation easier: (in terms of hProlog's suspension frames) a conjunction like *re_entry(Susp), re_entry(Susp)* requires special attention to the return address to be used on the *go_back*, if the body of the suspension can leave alternatives. We can take care of that at the same time as recursive activation. On the other hand, enforcing determinism is expensive. Often determinism can be detected because the body consists of arithmetic only, so that enforcing is it not necessary, but in the *nrev* benchmark - where detecting is beyond simple heuristics - enforcing determinism increases the time for the benchmark from 328 to 410 msec. That is still well below B-Prolog's timing, but it shows how big the cost can be.

5.3 Recursive activation

Recursive activation of a suspension can in principle occur for example in

```
p(X,Y), {ins(X), ins(Y)} =>
        Y = world,
        writeln(X+Y).
```

combined with the query *?- p(X,Y), X = hello*. The suspension frame is thereby visited twice and in a nested way.

However, B-Prolog seems to give a different execution order: the nested activation of the suspension is postponed until the current one is finished. We do not know whether this is intentional, or an implementation feature. The event/2 trigger can activate the same suspension in a nested way in B-Prolog and moreover, [10] explicitly mentions that nested activation of a suspension frame is implemented by copying the frame. We take one step back from the B-Prolog implementation: one can argue that nested calls should be forbidden or postponed, because action rules are meant for constraint propagation. and it makes sense not to nest propagators. We will just describe how we could deal with a number of alternatives.

- in any approach, it seems mandatory that one can detect easily whether a suspension frame is being used at any moment: if the (E,CP) slots in the

- frame are zero when the frame is not in use, and set to their needed values otherwise, these slots can be used; `go_back` should set the slots back to zero
- if one does not allow nested activation, one can
 - simply fail when a nested call is detected
 - or postpone the new activation until the current activation is finished
 The second alternative is a bit difficult in case execution can backtrack into a suspension clause - something B-Prolog prevents, but hProlog supports.
- if one wants to support nested activation, the (E,CP) slots in the frame must be managed like a stack; this is possible using lists and backtrackable destructive update (mutables) or by copying the suspension frame

Our implementation currently does not support nested activation: recursive activation simply fails. Backtracking into a suspension is fully supported.

5.4 Garbage collection

A few adaptations to the garbage collector are needed. We describe the issues during a marking phase:

- the suspension term was implemented as a special functor with arity 4, i.e. a slot for the E pointer which points to a suspension frame, a slot for CP, one Alive slot, and one Message slot (see Section 5.7); the latter two are ordinary Prolog terms and need no special treatment; if a suspension term is reachable from the root set, and the Alive slot indicates that the suspension frame is not yet killed, the suspension frame must be kept alive - see later
- a suspension frame can also be pointed to by a (E,CP) pair in an ordinary environment (i.e. on the local stack) or another suspension frame
- keeping a suspension frame alive means that as a whole it must be retained by the collector: the collector must be able to find out what its size is; in [4] we have described how we simply precede the environment on the heap with its size - as given in the `alloc_heap` instruction
- keeping a suspension frame alive means that its slots are added to the root set: the CP that comes with the suspension term, or the CP in the (E,CP) combination found elsewhere. is normally used to determine the amount of relevant slots (or the live variable map) and we can do the same here
- when a suspension term indicates that the suspension is killed, the suspension frame is garbage: in fact the suspension term itself can be considered garbage, but it is reachable and `re_entry/1` could still get it as an argument; still, it is important to be able to collect such garbage suspension terms; since the compiler generates the code that deals with suspension terms from higher level code that cannot see them, we can indeed do such a collection; for instance, if a single suspension term (not a conjunction) is attached to a live variable, but the suspension is killed, the garbage collector can safely remove the suspension term from the variable - conditionally trailing the removal of course; the situation with conjunctive suspensions is slightly more complicated and not described here in detail

5.5 Coroutines

Our source level approach to suspensions allows a more finegrained control. We need a new built-in `update_re_entry_point/1`. Here is a suggestive piece of code and query:

```
:- suspension(foo/3).
foo(Susp) :-
    re_entry_point(Susp),
    writeln(1),
    update_re_entry_point(Susp),
    writeln(2),
    go_back.

?- foo(Susp), re_entry(Susp), re_entry(Susp).
```

The idea is that the output of the query is **12**: `update_re_entry_point/1` updates the code pointer part of the suspension term. Once this facility is present, true (hard wired) concurrency between agents can be achieved.

5.6 Top of Stack

hProlog computes the Top of Stack pointer `tos` on deallocating an environment using the following code⁷:

```
ereg = ereg[0]; % deallocate environment
if (ereg > b[tos_slot]) tos = b[tos_slot];
else tos = ereg;
```

`Tos` must point in the local stack, and since `ereg[0]` could contain a pointer to a suspension frame on the heap, this code must be adapted. One correct adaptation would be:

```
ereg = ereg[0]; % deallocate environment
tmp = ereg;
while (onheap(tmp)) tmp = tmp[0]; % skip the suspension frames
if (tmp > b[tos_slot]) tos = b[tos_slot];
else tos = tmp;
```

However, a chain of suspension frames might need skipping and the complexity of the program can easily become worse.⁸ A better way is as follows: when allocating a suspension frame on the heap, save first the current `tos` on the heap, so that it can be retrieved easily. Adapt the code for deallocate to:

⁷ In hProlog, environments grow to lower addresses and `ereg` points to the lowest address in the environment.

⁸ We found out the hard way.

```

ereg = ereg[0]; % deallocate environment
tmp = ereg;
if (onheap(tmp)) tmp = tmp[-1]; % fetch saved tos
if (tmp > b[tos_slot]) tos = b[tos_slot];
else tos = tmp;

```

5.7 The event/2 trigger

We shortly indicate how the event/2 trigger can be supported by basically the same suspension frame mechanism: this is not a surprise of course, as B-Prolog does this already.

Our suspension terms had from the beginning an extra slot: the message slot. We use it in a few built-in predicates as follows: as an example we show how the following Action Rule code is compiled to hProlog.

```
p(X), {event(X,M)} => writeln(M).
```

is transformed to:

```

:- suspension(p/1).

p(X) :-
    attach_event(X,Susp),      %
    re_entry_point(Susp),     %
    get_message(Susp,M),      %
    writeln(M),
    go_back.

```

The lines with % contain a call to a new built-in:

- attach_event/2 works like attach_suspension, except that it treats a conjunction slightly different - we go in more detail later
- get_message/2 picks up the message from the suspension term

Suppose that a variable X is waiting for only one message (i.e. there is no conjunction), then posting an event is achieved by a call *post_event(X,hello)*. The code for post_event/2 is:

```

post_event(X,M) :-
    get_susp(X,Susp),      %
    set_message(Susp,M),  %
    re_entry(Susp).

```

The lines marked % perform the following: `get_susp` retrieves the suspension term from `X`; `set_message` sets the message slot of the suspension term to `M`.

Dealing with a conjunction of message suspensions can be achieved by the following code:

```
:- suspension(event_conj/3).

event_conj(Susp1,Susp2,Conj) :-
    re_entry_point(Conj),
    get_message(Conj,M),
    set_message(Susp1,M),
    re_entry(Susp1),
    set_message(Susp2,M),
    re_entry(Susp2),
    go_back.
```

Caveat The implementations of the `ins/2` trigger and the `event/2` trigger can at this point not cooperate, i.e. code like

```
p(X), {ins(X), event(X,M)} => writeln(M).
```

cannot be dealt with yet. This used to be an unsupported feature in earlier versions of B-Prolog as well and the current version gives it a seemingly ad-hoc meaning which we are not prepared to mimick without further understanding its semantics.

6 Alternatives

We discuss briefly some alternatives to the suspension on heap approach we have presented in this paper:

- It is easy to make a WAM which keeps **all** environments on the heap: this is viable and in [4] we have shown that performance does not suffer. It would seem that treatment of suspension frames becomes easier in such an approach. However, the issues mentioned in Section 5 would remain, so no great simplification of the implementation would result.
- B-Prolog uses copying on recursive activation of a suspension: this is only vaguely described in [10], but the idea seems to be that a full copy is made of the suspension frame. This would also do in our setting of course: we did it our way just to explore an alternative to what was done before.
- We could have copied the TOAM approach and put suspension frames on the local stack. This is clearly feasible, but we are not fond of that because it complicates the implementation by requiring a local stack collector. Even more, the other issues related to garbage collection, make us reluctant to trade in the WAM meta-call approach for a suspension frame approach that is not essentially more efficient.

7 Conclusion

Our main objective was to provide WAM implementors with a relatively easy to incorporate alternative to a meta-call approach for the re-activation of goals. This alternative was described in [9] and at that moment seemed very tightly connected to the overall design of the TOAM. We have shown that the creation of re-entrant suspension frames carries over to the WAM and that some implementation decisions have little effect on performance: whether suspension frames should be kept on the stack or on the heap can be decided on other criteria, e.g. the ease of integration with garbage collection, stack management etc. This paper also shows that the WAM and the TOAM realisation of the suspension frames idea have similar performance. This confirms our earlier findings, namely that the TOAM and the WAM offer essentially the same performance opportunities. It is up to implementors to exploit them.

Based on suspension frames, the Action Rule language can be implemented efficiently. Action Rules form a powerful tool for the constraint solver programmer, as shown in [10]. The efficient implementation of Action Rules seemed reserved to B-Prolog. This paper shows that any WAM based implementation can take advantage of the expressive power of Action Rules. Hopefully, this will have a positive impact on future developments of constraint solvers in Prolog.

Comparing the current work here with the work in [6], there is one technical difference that seems small, but has quite a big impact: in the spirit of suspensions on the heap, we have implemented conjunctions of suspensions by creating a new suspension. In [6] conjunctions are represented by a list. The latter choice is more flexible, and we are inclined to replace the conjunction suspension by such lists in our future work.

Acknowledgements

Part of this work was conducted while the first author was a guest at the Institut de Mathématiques Appliquées of the Université Catholique de l'Ouest in Angers, France. Sincere thanks for this hospitality. We also thank Henk Vandecasteele for maintaining the hipP compiler which we use within hProlog.

References

1. H. Aït-Kaci. *Warren's Abstract Machine: A Tutorial Reconstruction*. MIT Press, 1991.
2. B. Demoen. Dynamic attributes, their hProlog implementation, and a first evaluation. Report CW 350, Dept. of Computer Science, K.U.Leuven, Belgium, Oct. 2002.
3. B. Demoen and P.-L. Nguyen. So many WAM variations, so little time. In J. Lloyd, V. Dahl, U. Furbach, M. Kerber, K.-K. Lau, C. Palamidessi, L. M. Pereira, Y. Sagiv, and P. J. Stuckey, editors, *Computational Logic - CL2000, First International Conference, London, UK, July 2000, Proceedings*, volume 1861 of *Lecture Notes in Artificial Intelligence*, pages 1240–1254. ALP, Springer, 2000.

4. B. Demoen and P.-L. Nguyen. Allocating wam environments/choice points on the heap. Report CW 455, Dept. of Computer Science, K.U.Leuven, Belgium, July 2006.
5. B. Demoen and P.-L. Nguyen. Reconstructing a semantics for action rules by a transformation to almost plain prolog. Report CW 456, Department of Computer Science, K.U.Leuven, Leuven, Belgium, August 2006. URL = <http://www.cs.kuleuven.ac.be/publicaties/rapporten/cw/CW345.abs.html>.
6. B. Demoen and L. Nguyen-Phuong. Delay and events in the toam and the wam. In *CICLOPS 2006: Colloquium on Implementation of Constraint Logic Programming Systems*, pages 63–79, 2006.
7. D. H. D. Warren. An abstract Prolog instruction set. Technical report, SRI International, 1983.
8. N.-F. Zhou. Global optimizations in a Prolog compiler for the TOAM. *Journal of Logic Programming*, 15(4):275–294, 1993.
9. N.-F. Zhou. A Novel Implementation Method for Delay. In *Joint International Conference and Symposium on Logic Programming*, pages 97–111. MIT Press, 1996.
10. N.-F. Zhou. Programming finite-domain constraint propagators in action rules. *Theory and Practice of Logic Programming (TPLP)*, 6(5):483–508, 2006.
11. N.-F. Zhou and M. Wallace. A simple constraint solver in action rules for the cp'05 solver competition. In *Proceedings of the CP workshop on Constraint Propagation and Implementation*, 2005.

Appendix

The *real* benchmarks

Nrev

```
% Action Rules
concat(I,_,_), var(I), {ins(I)} => true.
concat([],I,0) => I = 0.
concat([X|R],I,Out) => Out = [X|00], concat(R,I,00).
```

```
% Freeze/2
concat(X,Y,Z) :- freeze(X,concat1(X,Y,Z)).

concat1([],L1,L2) :- L2=L1.
concat1([X|Xs],L1,L2) :- L2=[X|L3], concat(Xs,L1,L3).
```

```
% Suspension on heap
:- suspension(concat/3).
concat(X,Y,Z) :-
    (var(X) ->
        attach_suspension(X,L),
        re_entry_point(L),
        concat1(X,Y,Z),
        go_back
    ;
        concat1(X,Y,Z),
        go_back
    ).

concat1([],L1,L2) :- L2=L1.
concat1([X|Xs],L1,L2) :- L2=[X|L3], concat(Xs,L1,L3).
```

Sendmoremoney

```
% Action Rules
neq(X,_), var(X), {ins(X)} => true.
neq(X,Y) => X\=Y.

eq(X,Y), var(X), {ins(X),ins(Y)} => true.
eq(X,Y), var(Y), {ins(Y)} => true.
eq(X,Y) => X:=Y.

add(C,X,Y,NewC,Z),var(C),{ins(C),ins(X),ins(Y),ins(NewC),ins(Z)} => true.
add(C,X,Y,NewC,Z),var(X),{ins(X),ins(Y),ins(NewC),ins(Z)} => true.
add(C,X,Y,NewC,Z),var(Y),{ins(Y),ins(NewC),ins(Z)} => true.
add(C,X,Y,NewC,Z),var(NewC),{ins(NewC),ins(Z)} => true.
add(C,X,Y,NewC,Z),var(Z),{ins(Z)} => true.
add(C,X,Y,NewC,Z) => C+X+Y:=10*NewC+Z.
```

```
% Freeze/2
neq(X,Y) :- freeze(X,noteq(X,Y)).

noteq(X,Y) :- X\=Y.

eq(X,Y) :-
    Goal = equal(X,Y,_),
    freeze(X,Goal),
    freeze(Y,Goal).

equal(X,Y,Z) :-
    (nonvar(X), nonvar(Y), var(Z) ->
        X := Y,
        Z = 1
    ;
    true
    ).

add(C,X,Y,NewC,Z):-
    Goal = check(C,X,Y,NewC,Z,_),
    freeze(C,Goal), freeze(X,Goal), freeze(Y,Goal),
    freeze(NewC,Goal), freeze(Z,Goal).

ck(C,X,Y,NewC,Z,New) :-
    (nonvar(C),nonvar(X),nonvar(Y),nonvar(NewC),nonvar(Z),var(New) ->
        C+X+Y:=10*NewC+Z,
        New = 1
    ;
    true
    ).
```

```

% Suspension on heap
:- suspension(neq/2).
neq(X,Y) :-
    (var(X) ->
        attach_suspension(X,L),
        re_entry_point(L),
        X=\=Y,
        go_back
    ;
        X=\=Y,
        go_back
    ).

:- suspension(eq/2).
eq(X,Y) :-
    (var(X) -> attach_suspension(X,Susp), P = 1 ; true),
    (var(Y) -> attach_suspension(Y,Susp), P = 1 ; true),
    (var(P) ->
        X := Y,
        go_back
    ;
        re_entry_point(Susp),
        (
            nonvar(X), nonvar(Y) ->
            X := Y,
            kill_go_back(Susp)
        ;
            go_back
        )
    ).

:- suspension(add/5).
add(C,X,Y,NewC,Z) :-
    (var(C) -> attach_suspension(C,Susp), P = 1 ; true),
    (var(X) -> attach_suspension(X,Susp), P = 1 ; true),
    (var(Y) -> attach_suspension(Y,Susp), P = 1 ; true),
    (var(NewC) -> attach_suspension(NewC,Susp), P = 1 ; true),
    (var(Z) -> attach_suspension(Z,Susp), P = 1 ; true),
    (var(P) ->
        C+X+Y:=10*NewC+Z,
        go_back
    ;
        re_entry_point(Susp),
        (nonvar(Z), nonvar(NewC), nonvar(Y), nonvar(X), nonvar(C) ->
            C+X+Y:=10*NewC+Z,
            sysh:kill_go_back(Susp)
        ;
            go_back
        )
    ).

```

Sort

```
% Action Rules
smaller(X,Y), var(X), {ins(X), ins(Y)} => true.
smaller(X,Y), var(Y), {ins(Y)} => true.
smaller(X,Y) => X =< Y.
```

```
% Suspension on heap
:- suspension(smaller/2).
smaller(X,Y) :-
    (var(X) ->
        (var(Y) ->
            attach_suspension(X,LXY), attach_suspension(Y,LXY),
            re_entry_point(LXY),
            (nonvar(X), nonvar(Y) ->
                X =< Y, kill_go_back(LXY)
            );
            go_back
        )
    );
    attach_suspension(X,LX),
    re_entry_point(LX),
    X =< Y, go_back
);
    (var(Y) ->
        attach_suspension(Y,LY),
        re_entry_point(LY),
        X =< Y, go_back
    );
    X =< Y, go_back
).
```

```
% Freeze/2
sorted([X]):-!.
sorted([X,Y|L]):-
    Goal = smaller(X,Y,_), freeze(X,Goal), freeze(Y,Goal),
    sorted([Y|L]).

smaller(X,Y,Z) :-
    (nonvar(X), nonvar(Y), var(Z) ->
        X =< Y,
        Z = 1
    );
    true
).
```

Queens

```
% Action Rules
noattack(X,Y,K), var(X), {ins(X),ins(Y)} => true.
noattack(X,Y,K), var(Y), {ins(Y)} => true.
noattack(X,Y,K) =>
    X =\= Y,
    X+K =\= Y,
    X-K =\= Y.
```

```
% Freeze/2
safe(X,[Y|T],K):-
    Goal = noattack(X,Y,K,_),
    freeze(X,Goal),
    freeze(Y,Goal),
    K1 is K+1,
    safe(X,T,K1).

noattack(X,Y,K,New) :-
    (nonvar(X), nonvar(Y), var(New) ->
        X =\= Y,
        X+K =\= Y,
        X-K =\= Y,
        New = 1
    ;
    true
    ).
```

```
% Suspension on heap
:- suspension(noattack/3).
noattack(X,Y,K) :-
    (var(X) -> attach_suspension(X,LXY), P = 1 ; true),
    (var(Y) -> attach_suspension(Y,LXY), P = 1 ; true),
    (var(P) ->
        noattack1(X,Y,K),
        sysh:go_back
    ;
        sysh:re_entry_point(LXY),
        (nonvar(X), nonvar(Y) ->
            noattack1(X,Y,K),
            sysh:kill_go_back(LXY)
        ;
            sysh:go_back
        )
    ).

noattack1(X,Y,K) :- X =\= Y, X+K =\= Y, X-K =\= Y.
```