Combining an Improvement to PARMA Trailing with Trailing Analysis

Tom Schrijvers
Department of Computer Science
Katholieke Universiteit Leuven
Celestijnenlaan 200A
B-3001 Heverlee, Belgium
tom.schrijvers@cs.kuleuven.ac.be

Bart Demoen
Department of Computer Science
Katholieke Universiteit Leuven
Celestijnenlaan 200A
B-3001 Heverlee, Belgium
bart.demoen@cs.kuleuven.ac.be

ABSTRACT

Trailing of bindings in the PARMA variable representation is expensive in time and space. Two schemes are presented that lower its cost: the first is a technique that halves the space cost of trailing in PARMA. It can be used with conditional and unconditional trailing. It is illustrated and evaluated in the context of dProlog and in the Mercury backend of HAL. The second scheme combines a variant of a previously developed trailing analysis with the first technique. Empirical evidence shows the usefulness of these schemes and that the combination is more effective than each scheme apart.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors—*Compilers,Optimisation*

General Terms

Performance

Keywords

PARMA, trailing, program analysis

1. INTRODUCTION

We will assume working knowledge of Prolog and its implementation without further explanation. For a good introduction to Prolog see [15]; to the WAM see [1, 22].

In the WAM, a free non-aliased variable is represented as a cell containing a self-reference. When two free variables are unified the younger cell is made to point to the older cell. A series of unifications of free variables can result in a *linear* chain of references of which the last one is a self-reference or, in case the variable is instantiated, a bound term. This implies that testing whether a (source level) variable is free or bound, requires dereferencing. Such dereferencing is necessary during each unification and thus performed quite often.

In his PARMA-system [19], Taylor introduced a different variable representation scheme that does not suffer from this massive dereferencing need. When in the PARMA scheme two free non-aliased variables are unified, the two involved cells are made to point to each other, creating a *circular* chain of length two. Further unifications with free variables make this circular chain larger and every variable (in the local stack or an argument register) points to a cell in such a chain. When the variable is bound, each cell in the PARMA chain is replaced by the value to which it is bound. No dereferencing is required to verify whether a cell is bound, because the tag in a cell immediately identifies a cell as being bound or not. A good account of the advantages and disadvantages of the PARMA binding scheme can be found in [8].

Although the PARMA scheme avoids dereferencing in boundness checks, this improvement over the WAM scheme has its price. When during a unification a cell is assigned a value, trailing is used in order to be able to reconstruct the state of the abstract machine before the unification took place. In the case of the WAM, at most one cell needs trailing, regardless of the length of the linear chain. On the other hand, in the PARMA scheme, every cell in the circular chain potentially needs trailing. As a result, the trail stack usage is expected to be much higher in the PARMA scheme than in the WAM. Demoen and Nguyen [3] have indeed observed in the dProlog system maximal trail sizes for the PARMA scheme that are on average twice as large as with the WAM scheme. Even in the WAM it pays off to avoid trailing whenever possible and a fortiori this holds in the PARMA scheme.

The PARMA scheme is used in HAL: HAL [5, 4] is a constraint logic language designed to support the construction, extension and use of constraint solvers. HAL requires type declarations and has optional mode and determinism declarations. It is compiled to Mercury [16] so as to leverage from its sophisticated compilation techniques. However, unlike Mercury, HAL includes a Herbrand constraint solver which provides full unification. This solver uses Taylor's PARMA scheme rather than the standard WAM representation. This is because, unlike the WAM, the PARMA representation for ground terms is the same as that of Mercury in a very important aspect: no reference chains exist. Thus, for ground terms calls to the Herbrand constraint solver can be replaced by calls to Mercury's efficient routines.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
PPDP'02, October 6-8, 2002, Pittsburgh, Pennsylvania, USA.
Copyright 2002 ACM 1-58113-528-9/02/0010 ...$5.00.

We will present two means to counter the trailing penalty of the PARMA scheme. The first is an improved PARMA trailing scheme, implemented both in dProlog and the HAL-Mercury system: it considerably reduces the required trail stack size. It is presented in Section 2. On top of this improved trailing scheme for HAL-Mercury we have implemented an analysis that detects when trailings can be avoided. This analysis is similar to the one we have previously implemented for the classic PARMA trailing scheme [13], but takes into account the particular properties of the improved trailing scheme. Section 3 explains the analysis which attempts to avoid trailing in the improved scheme. Results of both the improved scheme and the improved scheme in combination with the analysis are presented in Section 4. Finally, Section 5 discusses related and future work.

2. IMPROVING THE TRAILING SCHEME

2.1 Conditional versus unconditional trailing

A cell that is changed, only requires trailing if the cell did exist before the most recent choice point. Otherwise there is no previous state that has to be restored after backtracking. This property applies equally in the WAM and the PARMA scheme.

In some systems a simple runtime test can be used to verify whether a cell is older than the most recent choice point. If all cells on the heap are kept in order of allocation, the address of the cell can simply be compared to `bh`, the address of the top of the heap at the beginning of the most recent choice point. In this kind of systems, like for instance dProlog, conditional trailing is used: the runtime test is performed to decide if the trailing is actually required.

In systems where the order of cells on the heap is not guaranteed unconditional trailing is required. The HAL-Mercury system for example is a system with unconditional trailing: it uses the Boehm garbage collector that does not preserve the order of the cells on the heap between garbage collections. Also other systems use unconditional trailing, at least during some unifications (see for instance [21]).

In [3] it is shown that global performance is hardly affected by the choice between conditional or unconditional trailing. The savings made on avoided trailings are balanced by the overhead of the runtime test.

In the following we will specify trailing operations for both conditional and unconditional trailing.

2.2 The classic PARMA Scheme: Value trailing

The classic PARMA trailing scheme uses *value trailing*, described by the following C-like code[1]:

```
valuetrail(p, tr) {
      *(tr++) = *p;
      *(tr++) = set_tag(p,VALUE_TRAIL);
}
```

In this code `p` is the address of a cell in a PARMA chain, `tr` points to the top of the trail stack. The above trailing is unconditional. It stores both the address and the value of a cell. Tagging of the last item pushed on the trail stack is necessary to distinguish between the different kinds of trailing information, as is explained in Section 2.5.

The untrail operation for value trailing is straightforward:

```
untrail_valuetrail(tr) {
      address = untag(*(tr--));
      *address = *(tr--);
}
```

The variant for conditional trailing includes the aforementioned runtime test:

```
cond_valuetrail(p, tr, bh) {
   if (p < bh) {
      *(tr++) = *p;
      *(tr++) = set_tag(p,VALUE_TRAIL);
   }
}
```

We will next show that the trail stack usage of value trailing can be greatly improved. The discussion distinguishes two case: variable–variable unification and variable–non-variable unification.

2.3 The improved trailing scheme for Variable–variable unification

Variable–variable unification results in the merging of the chains of the two involved variables. This merging is obtained by swapping the successors of the cells that the two variables point to. This swapping only changes those two cells, hence they are the only ones that must be trailed. In the classic scheme the value trailing of both cells takes up four trail stack slots when trailing is unconditional.

Undoing such variable–variable unification consists of restoring the old values of the cells separately. However, there is a more economic inverse operation that undoes the swapping that happened during unification: simply swapping back. This swapping only requires the addresses of the involved cells and not their respective old contents. We introduce a new kind of trailing named *swap trailing* which exploits this and also the corresponding untrailing operation. Swap trailing is defined by the following code:

```
swaptrail(p, q, tr) {
      *(tr++) = p;
      *(tr++) = set_tag(q,SWAP_TRAIL);
}
```

Here `p` and `q` are the addresses of the two cells. Swap trailing only consumes two slots in the trail stack. The untrail operation for swap trailing is:

```
untrail_swaptrail(tr) {
      q = untag(*(tr--));
      p = *(tr--);
      tmp = *q;
      *q = *p;
      *p = tmp;
}
```

The above describes the case where both cells are unconditionally trailed. In the conditional value trailing case, the classic scheme would either consume zero, two or four slots if respectively none, only one or both variables are older than the most recent choice point. Swap trailing can only be used there to replace the four slot case. Value trailing is still used for the two slot case. As a result the code for conditional variable–variable trailing looks like:

[1]All code in this paper is simplified pseudo-code. Implementation details that rather obfuscate than clarify the concepts at hand, have been left out.

```
cond_varvartrail(p, q, tr, bh) {
      if (p < bh) {
        if (q < bh) {
           swaptrail(p,q,tr);
        } else {
           valuetrail(p, tr);
        }
      } else if (q < bh) {
         valuetrail(q, tr);
      }
}
```

It is clear that the potential gain in space on the trail comes at a cost in time and that the gain in space is not guaranteed.

2.4 The improved trailing scheme for Variable–nonvariable unification

Variable-nonvariable unification pulls the entire chain of the variable apart. Every cell in the chain is set to the nonvariable. As every cell is changed in this way, every cell has to be (conditionally) trailed in order to be able to reconstruct the chain during backtracking. If the chain consists of n cells, then the unconditional value trailing consumes $2n$ slots on the trail stack. Conditional trailing consumes $2k$ cells if k of the n cells are older than the most recent choice point.

In the case of unconditional value trailing, every address of a cell is stored twice: once as the address of a cell and once as the contents of the predecessor cell. This means that there is quite some redundancy. The obvious improvement is to store each address only once. We name this *chain trailing*. Because the length of the chain is not known, a marker is needed to indicate, for the untrailing operation, where chain trailing ends. The last entry of the chain encountered during untrailing, is the first one actually trailed. We use the `CHAIN_END` tag to mark this entry.

The last address put on the trail is tagged with `CHAIN_BEGIN` to indicate the kind of trailing. For chains of length one, the last and first cell coincide. The `CHAIN_END` tag is used to mark this single address.

Chain trailing is defined by the code:

```
chaintrail(p, tr) {
   start = p;
   *(tr++) = set_tag(p,CHAIN_END);
   p = *p;
   only_one = TRUE;
   while (p != start) {
      only_one = FALSE;
      *(tr++) = p;
      p = *p;
   }
   if (!only_one) {
      last = tr - 1;
      *last =
         set_tag(*last,CHAIN_BEGIN);
   }
}
```

The untrail operation for reconstructing the chain is straightforward: it dispatches to the appropriate untrailing action depending on the tag of the first cell encountered during untrailing. In case this is `CHAIN_BEGIN`, meaning $n \geq 2$, the corresponding code is:

```
untrail_chaintrail(tr) {
   head = untag(*(tr--));
   previous = head;
   current = *(tr--);
   while (get_tag(current)
               != CHAIN_END) {
     *current = previous;
     previous = current;
     current = *(tr--);
   }
   current = untag(current);
   *current = previous;
   *head = current;
}
```

In case the first tag encountered is `CHAIN_END`, $n = 1$ and this code for untrailing is:

```
untrail_shortchain(tr) {
   cell = untag(*(tr--));
   *cell = cell;
}
```

If $2 * k < n$, unconditional chain trailing consumes more space than conditional value trailing. But a conditional variant of chain trailing is also possible:

```
cond_chaintrail(p, tr, bh) {
   start = p;
   first = TRUE;
   only_one = TRUE;
   do {
      if (p < bh)
         if (first) {
            *(tr++) =
               set_tag(p,CHAIN_END);
            first = FALSE;
         } else {
            only_one = FALSE;
            *(tr++) = p;
         }
      p = *p;
   } while (p != start);
   if (!first && !only_one) {
      last = tr - 1;
      *last =
         set_tag(*last,CHAIN_BEGIN);
   }
}
```

This conditional variant uses only k slots of the stack trail, so it is clearly an improvement over conditional value trailing whenever $k > 0$.

The untrail operation used is the same as for the unconditional chain trailing. This looks weird at first: the cond_chaintrail has not put all the cells of the chain on the trail. The cells younger than the most recent choice point are indeed not on the trail. So one can wonder whether cond_chaintrail leaves enough information on the trail in order to reconstruct the chain that existed just before the binding was made. However, that is not actually required. The objective of trailing is to be able to reconstruct the bindings that existed at the creation time of a choice point; intermediate states during untrailing need not be consistent.

By considering how k could be smaller than n it becomes quite clear that the conditional chain trailing together with

previous trailings since the most recent choice point makes it possible to reconstruct the situation at the creation time of that choice point. The $n - k$ cells that are not trailed in the chain trailing belong to variables younger than the most recent choice point. They must have been introduced in the chain by means of a variable–variable unification earlier on. When this variable–variable unification involved an older cell that was value trailed. Then the inconsistent chain, obtained by untrailing the chain trailing, is brought back to its consistent state at the beginning of the choice point by untrailing the value trailing.

In fact the more general – and better with respect to stack trail consumption – principle behind this is that only the old (older than the most recent choice point) cells in the chain pointing to other old cells have to be trailed. The kind of trailing suitable for this is a special kind of value trailing, where the successive equal slots on the trail stack are overlapped. The above cond_chaintrail only approximates this, as an implementation would incur an undue time overhead because of the extra runtime tests: we also store the addresses of old cells neither pointing to or pointed to by old cells.

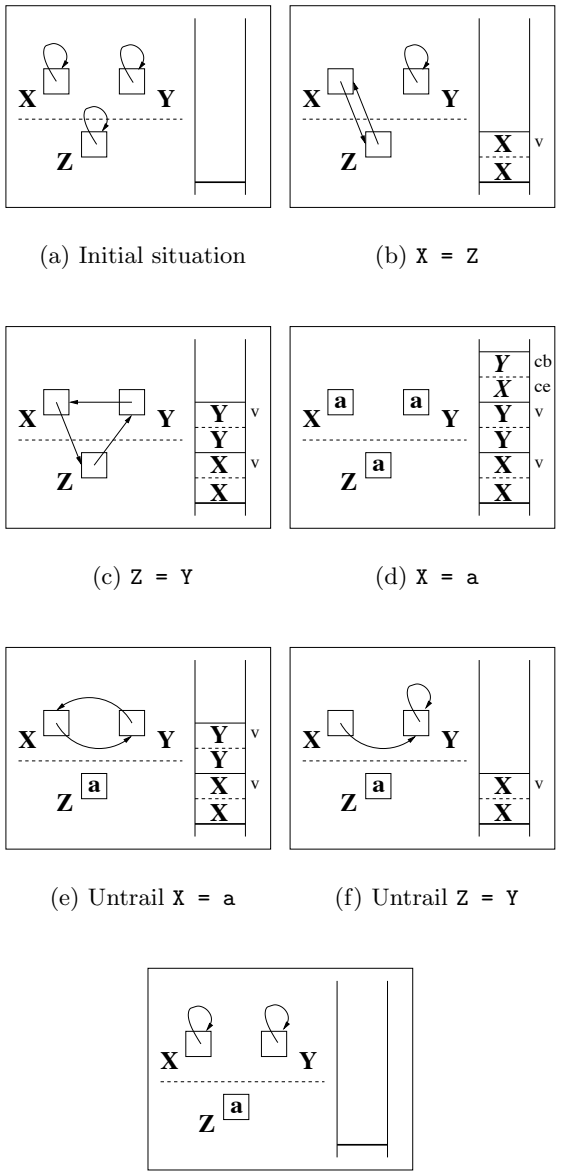
Example.
Figure 1 illustrates with a small example how the above specified conditional chain trailing together with previous trailings safely restores the state of all variables older than the most recent choice point:

```
X = Z,
Z = Y,
X = a,
fail
```

Suppose that `X` and `Y` are older than the most recent choice point, `Z` is newer and all three are chains of length 1 as depicted in Figure 1(a). The successive forward steps are shown in the Figures 1(b), 1(c) and 1(d). X is value trailed during `X = Z`, as is `Y` during `Z = Y`. The addresses of `X` and Y are stored on the trail stack with conditional chain trailing during `X = a`[2].

The `v`, `cb` and `ce` to the side of the stack trail entries represent the `VALUE_TRAIL`, `CHAIN_BEGIN` and `CHAIN_END` tags respectively.

The execution fails immediately after `X = a`, and backtracks to the initial state in three steps. First (see Figure 1(e)) the conditional chain trailing is untrailed, creating a chain of X and Y. Next (see Figure 1(f)) the value trailing of Y is undone and finally (see Figure 1(g)) the value trailing of X is reversed too. The final state corresponds to the initial state, except for Z, which is still bound to a. However as Z did not exist before the most recent choice point, its content is irrelevant at that point because it is inaccessible and will have been reclaimed from the heap anyway when forward execution resumes.



(a) Initial situation (b) `X = Z`

(c) `Z = Y` (d) `X = a`

(e) Untrail `X = a` (f) Untrail `Z = Y`

(g) Untrail `X = Z`

Figure 1: Conditional chain trailing example.

2.5 Combining the improvements

In the improved unconditional trailing scheme of HAL-Mercury, in addition to swap and unconditional chain trailing, also function trailing is used to allow custom trailings for constraint solvers. Function trailing stores a pointer to an untrailing function and to untrailing data. The untrailing process must be able to distinguish between the different kinds of trailing information on the trail stack so that the appropriate untrail operation can be called. As said previously, a simple tagging scheme is used to indicate the kind of trailing information. Fortunately there are two tag bits available[3] and that is just enough to distinguish between the four kinds of trailing information. There is one constraint on the allocation of the four different tags to the kinds of trailing: the `CHAIN_END` tag should not look the same as the tag of the intermediate addresses in a chain trail.

The general untrail operation then simply looks like:

[2]this looks superfluous, but in the absence of information on what has been trailed before, it is unavoidable.

[3]because of the alignment

```
untrail(tr,tr_cp) {
   while (tr > tr_cp) {
      switch (get_tag(*tr)) {
         case FUNCTION_TRAIL:
            untrail_functiontrail(tr);
            break;
         case SWAP_TRAIL:
            untrail_swaptrail(tr);
            break;
         case CHAIN_BEGIN:
            untrail_chaintrail(tr);
            break;
         case CHAIN_END:
            untrail_shortchain(tr);
      }
   }
}
```

In the improved conditional trailing scheme of dProlog only value, swap and conditional chain trailing are used. The remarks on the application and allocation of tags is the same as for the unconditional case and the general conditional untrail operation looks similar:

```
cond_untrail(tr,tr_cp) {
   while (tr > tr_cp) {
      switch (get_tag(*tr)) {
         case VALUE_TRAIL:
            untrail_valuetrail(tr);
            break;
         case SWAP_TRAIL:
            untrail_swaptrail(tr);
            break;
         case CHAIN_BEGIN:
            untrail_chaintrail(tr);
            break;
         case CHAIN_END:
            untrail_shortchain(tr);
      }
   }
}
```

When looking at the value trailings of chains of length one in the example in the previous section (see Figure 1), there is an obvious trailing alternative in the conditional system that stores no redundant information: chain trailing. Indeed if such a variable would be chain trailed instead of value trailed, only one instead of two slots would be used on the stack. However this would require more runtime tests and we have not implemented this.

Experimental results for both the conditional and unconditional trailing scheme are presented in Section 4.

3. TRAILING ANALYSIS

The object of a trailing analysis is to find unifications in a program that do not require any trailing of one or more involved variables. This analysis information then can be used to replace those particular unifications with more efficient variants so that both execution time and trail stack usage improve.

The trailing analysis heavily depends on the details of the trailing scheme. In [13] we have presented a trailing analysis for the classic PARMA trailing scheme. In comparison to that scheme it will become clear that because of its stack trail economy the improved trailing scheme gives rise to less opportunities for trailing elimination for further stack trail savings.

3.1 Unnecessary trailing in the improved trailing scheme

The need for trailing arises from choice points and backtracking. Once a certain path in a program has been fully explored backtracking brings the execution back to the most recent choice point to continue along another path. During backtracking information on the trail stack is used to reconstruct the state of the program, its variables, to what it was at the time of the choice point.

Only at choice points should the state be reconstructable. Value trailing, the only kind of trailing of the classic scheme, however is a much more powerful means of trailing, as it allows for the reconstruction of a cell to its state at the time of trailing. If a cell is value trailed several times after a choice point (before any later choice point), then only the earliest of those trailings is needed. Further value trailings in addition allow for reconstruction of a cell at an intermediate point. As those intermediate states are useless[4], it is evident that a trailing analysis for the classic scheme has a lot of opportunity to detect spurious trailings of this kind.

If we consider the improved scheme it becomes clear that swap trailing, as opposed to value and chain trailing, is an incremental kind of trailing, relying on future trailings for proper untrailing of cells. During the untrailing process all later chain and swap trailings have to be undone before the swap trailing can be untrailed correctly. Essentially this is because the content of the cells is not stored during the trailing, but only the incremental change. So there is no opportunity here to avoid any future trailings after the first one between two choice points.

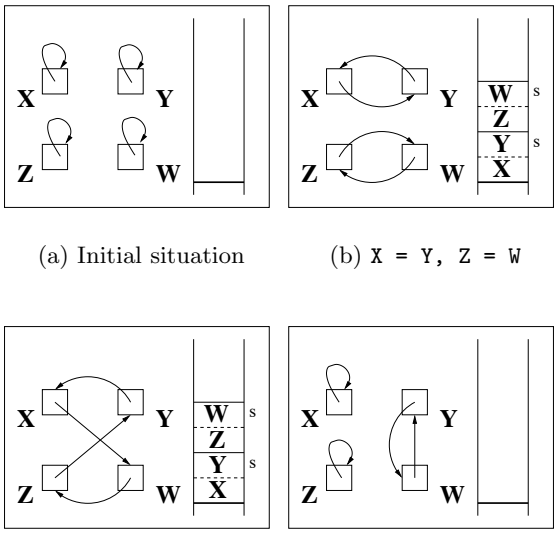
Counterexample.
Consider this small counterexample (see Figure 2) where we wrongly assume that a variable needs not be trailed a second time between two choice points:

```
X = Y,
Z = W,
X = Z,
fail
```

Initially, all variables are represented as chains of length one, as depicted in Figure 2(a). All variables are older than the most recent choice point. In the first two steps the four variables are aliased and swap trailed pairwise, creating two chains of length two (see Figure 2(b)). The `s`'s represent `SWAP_TRAIL` tags.

Next X and Z are aliased, creating one large chain (see Figure 2(c)). During this step X and Z are not (swap) trailed since they have been swap trailed before since the most recent choice point - and because of the wrong assumption. Finally the execution fails and untrailing tries to restore the situation at the most recent choice point. However Figure 2(d) shows that the omission of the last swap trailing was invalid, as untrailing fails to restore the correct situation.

[4]This is assuming that the semantics of function trailing are such that they do not rely on the intermediate state of any Herbrand variables during untrailing.



(a) Initial situation (b) `X = Y, Z = W`

(c) `X = Z` (d) `Untrail`

Figure 2: Example of incremental behaviour of swap trailing: it does not eliminate the need for further trailing of the same cells.

This is the big difference with the analysis for the classic trailing scheme: a cell involved in swap trailing still needs trailing later in the same segment of the execution.

Still there remains one opportunity to avoid trailing, common to all trailing schemes. The improved conditional scheme already detects this case at runtime: when a cell is more recent than the last choice point it is not going to be there after backtracking so it does not need to be trailed.

For non-heap order preserving systems however there is no easy runtime test to avoid this trailing. Fortunately this is where the trailing analysis can play its role: find the unifications of variables that have been initialised after the most recent choice point.

3.2 The analysis domain

The instantiation information obtained by HAL's mode analysis is available at each program point p as a table assigning to each variable in scope its instantiation. All HAL instantiations have an associated state which can be either `new`, `ground` or `old`. Instantiations with state `new` correspond to variables with no internal representation. Instantiations with state `ground` correspond to variables which are known to be bound to ground terms. In any other case the instantiations have state `old`, corresponding to variables which might not be bound but do have a representation (a chain of length one or more) or bound to a term not known to be ground. Variables assigned to instantiation with state `new`, `ground` or `old` will be called new, ground or old variables, respectively. Note that once a new variable becomes old or ground, it can never become new again. And once a variable is known to be ground, it can safely remain ground. Thus, the three states can be considered mutually exclusive. Let Var_p denote the set of all program variables in scope at p. A lookup in the instantiation table will be represented by the function $inst_p : Var_p \rightarrow \{new, ground, old\}$. This function allows us to partition Var_p into three disjoint sets: New_p, $Ground_p$ and Old_p containing the set of new, ground and old variables, respectively.

Similarly to [13] the descriptions – the elements of the trailing domain $L_{\texttt{trail}}$ – are partitions of only the old variables, those that could be represented by a PARMA chain or by structures containing PARMA chains. The variables are partitioned in only two groups here however: those that have to be trailed and those that don't. It is sufficient to only keep track of those variables that don't have to be trailed. Thus a description consists of the old variables at a program point that do not have to be trailed. We will call the property of an old variable, whether it has to be trailed or not, the trailing state of that variable in a particular description. The behaviour of the variables that don't have to be trailed is similar to that of the deep trailed variables in [13]. There are no shallow trailed variables in the current analysis, because swap trailing (which does apply to shallow trailed variables) does not obliterate the need for trailing of cells, as shown in Figure 2.

We will use l_p to denote an element of $L_{\texttt{trail}}$ at a particular program point p and Old_p for the set of `old` variables in scope at that program point.

Bound variables.
Bound variable are represented by a structure that potentially contains more than one chain. To simplify things the trailing state of all those chains is compacted into one single trailing state for the entire bound variable. Thereby, for safety, all chains have to be represented by the worst trailing state of any of the chains.

Consistency.
Trailing does not apply to variables but to chains of cells. However those chains are trailed through a particular variable. If it is not safe to trail some cells through one variable, than it should not be safe to trail them through any other variable. Hence all variables that share, i.e. potentially have one or more chains in common, should have the same trailing state. If at least one requires trailing, all other variables that share with it, should require trailing too.

We will use the information provided at each program point p by HAL's sharing analysis to define the function $share_p : Old_p \rightarrow \mathcal{P}(Old_p)$, which assigns to each variable in Old_p a set of variables in Old_p that possibly share with it. This information will be used to define the following function which makes trailing information consistent with its associated sharing information:

$$consist_p(l_p) = \{X \in l_p | share_p(X) \subseteq l_p\}$$

Lattice.
Define the partial ordering on $L_{\texttt{trail}}$ as $\supseteq$. Descriptions should be compared at the same program point only so that mode and sharing information are the same. Clearly $\langle L_{\texttt{trail}}, \supseteq \rangle$ is a complete lattice with top description $\top = \emptyset$ and bottom description $\bot = Old$.

3.3 Analysing HAL body constructs

This section defines the operations required by HAL's analysis framework [10] to analyse the different body constructs.

Variable-variable unification: $X = Y$**.**
There are several cases to consider:

- If either of the variables is `ground`, the other one will also be ground after unification. Neither will appear in the post-description.
- If one of the variables is `new`, then it will simply be a assigned a copy of the tagged pointer of the other variable. In the post-description its trailing state then should be the same as that of the other variable.
- If both variables need no trailing, then unification does not change this. As the merging of chains in the process does not introduce any cells that need to be trailed.
- If one variable, say X, needs no trailing and the other, Y, does, then after unification both need trailing. If the chains of both variables are merged then the chain of X will contain all the cells of Y's chain. Those had to be trailed. So now X has to be trailed to ensure that the cells of Y are trailed.

Formally, let l_1 be the pre-description and g_p be the set of ground variables at program point p. Its post-description l_2 can be obtained as:

$$
\begin{array}{l}
l_2 = \textbf{case } inst(X) \textbf{ of} \\
\quad new \;\rightarrow\; same(X, Y, l_1) \\
\quad old \;\rightarrow\; \textbf{case } inst(Y) \textbf{ of} \\
\qquad\qquad new \;\rightarrow\; same(Y, X, l_1) \\
\qquad\qquad old \;\rightarrow\; min(X, Y, l_1) \\
\qquad\qquad ground \;\rightarrow\; remove_ground(l_1, g_2) \\
\quad ground \;\rightarrow\; remove_ground(l_1, g_2)
\end{array}
$$

with the following definitions:

$$
\begin{array}{l}
remove_ground(l_i, v_i) = l_i \setminus v_i \\
same(X, Y, l_i) = \begin{cases} l_i \cup \{X\} & , Y \in l_i \\ l_i & , otherwise \end{cases} \\
min(X, Y, l_i) = \begin{cases} l_i & , \{X, Y\} \subseteq l_i \\ consist_2(l_i \setminus \{X, Y\}) & , otherwise \end{cases}
\end{array}
$$

Variable–term unification: $Y = f(X_1, \ldots, X_n)$
There are two cases to consider: If Y is new then the unification simply constructs the term in Y. Otherwise, the term is constructed in a fresh new variable Y' and the unification $Y' = Y$ is executed next. Since unifications of the form $Y' = Y$ have been discussed above, here we only focus on the construction into a new variable.

After the unification all involved variables will share with each other. Thus, since variables that do not have to be trailed can only share with other such variables, all involved variables should be removed in the post-description, if at least one of them was not in the pre-description.

Formally, let l_1 be the pre-description of the unification and x be the set of variables $\{X_1, \ldots, X_n\}$. Its post-description l_2 can be obtained as:

$$
l_2 = \begin{cases} l_1 \cup \{Y\} & , x \subseteq l_1 \\ consist_2(l_1 \setminus x) & , otherwise \end{cases}
$$

Predicate call: $p(X_1, \ldots, X_n)$**.**
Let l_1 be the pre-description of the predicate call and x the set of variables $\{X_1, \ldots, X_n\}$. The first step will be to project l_1 onto x resulting in description l_{proj}. Note that onto-projection is trivially defined as:

$$
onto_proj(l, v) = l \cap v
$$

The second step consists of extending l_{proj} onto the set of variables local to the predicate call. Since these variables are known to be new (and thus they do not appear in Old_1), the extension operation in our domain is trivially defined as the identity. Thus, from now on we will simply disregard the extension steps required by HAL's analysis framework.

The next step depends on whether the predicate is defined by the current module or by another (imported) module. Let us assume the predicate is defined by the current module and let l_{answer} be the answer description resulting from analysing the predicate's definition for calling description l_{proj}. In order to obtain the post-description, we will make use of the information provided by HAL's determinism analysis, i.e., information regarding the minimal-maximal number of solutions for each predicate. Like Mercury, HAL has six main kinds of determinism: `semidet` (0-1), `det` (1-1), `multi` (1-∞), `nondet` (0-∞), `erroneous` (1,0), `failure` (0-0). The determinism information is available as a table assigning to each predicate (procedure to be more precise) its inferred determinism. Thus, the post-description l_2 can be derived by combining the l_{answer} and l_1, using the determinism of the predicate call as follows:

- If the determinism is `multi` or `nondet`, then l_2 is equal to l_{answer} except for the fact that we have to apply the *consist* function in order to take into account the changes in sharing. This means that all variables that are not arguments of the call, should be trailed.
- Otherwise, l_2 is the result of combining l_{answer} and l_1: the trailing state of variables in x is taken from l_{answer}, while that of other variables is taken from l_1. The *consist* function has to be applied as well to take sharing between argument and non-arguments into account.

Formalised, the combination[5] function is defined as:

$$
\begin{array}{rcl}
l_2 & = & comb(l_1, l_{answer}) \\
 & = & \begin{cases} consist_2(l_{answer}) & , \texttt{multi } or \texttt{ nondet} \\ consist_2((l_1 \setminus x) \cup l_{answer}) & , otherwise \end{cases}
\end{array}
$$

Now, if the predicate is defined in an imported module, we will use the analysis registry created by HAL for every exported predicate: a table containing all call-answer description pairs encountered during analysis. Thus, we do a simple look up in this table and check if the predicate has a call-answer pair with a call description equal to l_{proj}. If there is no such description, then we will choose the smallest call-description that is less precise than l_{proj}. Since the table *always* includes a pair with the most general description ($\top$) as calling description, this selection process always

[5]Note that the combination is not the meet of the two descriptions. It is the "specialised combination" introduced in [12] which assumes that l_{answer} contains the most accurate information about the variables in x, the role of the combination being just to propagate this information to the rest of variables in the clause.

finds an appropriate variant. Finally, we combine the answer description l_{answer} of this call-answer pair with l_1 in the same way as for the intramodule call. HAL built-ins are treated in a similar way: a table is available containing the call-answer pairs for every built-in predicate.

Disjunction: $(G_1; G_2; \ldots; G_n)$**.**
Disjunction is the reason why trailing becomes necessary, because trailing might be needed for all variables which were already old before the disjunction. Thus, let l_0 be the pre-description of the entire disjunction. Then, $\top$ will be the pre-description of each G_i except for G_n whose pre-description is simply l_0 (since the disjunction implies no backtracking over the last branch).

Let $l_i, 1 \leq i \leq n$, be the post-description of goal G_i. We will assume that the set v_i of variables local to each G_i has already been projected out from l_i. The end result l_{n+1} of the disjunction is the least upper bound (lub) of all branches, which is defined as:

$$l_1 \sqcup \ldots \sqcup l_n = consist_{n+1}(remove_ground(l, g_{n+1}))$$

where

$$\begin{array}{rcl} l & = & l'_1 \cap \ldots \cap l'_n \\ l'_i & = & l_i \cup g'_i \end{array}$$

If-then-else: $I \rightarrow T ; E$ **.**
Although the if-then-else construct could be treated as $(I, T; E)$ this would be less accurate than needed since the determinism of I can be used to improve the accuracy. There are three cases to distinguish:

- I **has at least one solution,** i.e., the determinism is either `det` or `multi`[6]. This means that the E branch will never be executed. This case is thus equivalent to goal (I, T). The actual code transformation to (I, T) is not done in the HAL compiler, but in the Mercury compiler.

- I **has no solution, but simply fails,** i.e., the determinism is `failure`. Hence T will never be executed and this case is thus equivalent to goal E.

- **Otherwise.** The determinism is `semidet` or `nondet` and both branches might be executed. We still do not have to treat the if-then-else as a disjunction because we know that if one branch is executed, the other will not. Hence there is no explicit backtracking because of the if-then-else.

  Let l_1 be the pre-description to the if-then-else. Then l_1 will also be the pre-description to both I and E. Let l_I be the post-description obtained for I. Then l_I will also be the pre-description of T. Finally, let l_T and l_E be the post-descriptions obtained for T and E, respectively. Then, the post-description for the if-then-else can be obtained as the lub $l_T \sqcup l_E$.

  Note that this operation assumes a Mercury-like semantics of the if-then-else: No variable that exists before the if-then-else should be bound or aliased in such a way that trailing is required for backtracking if the condition fails. This is not a harsh restriction, since it is ensured whenever the if-condition is used in a logical way, i.e., it simply inspects existing variables and does not change any non-local variable.

[6]Actually, `erroneous` can also be treated like this.

Higher-order term construction: $Y = p(X_1, \ldots, X_n)$**.**
This involves the creation of a partially evaluated predicate, i.e., we are assuming there is a predicate with name p and arity equal or higher than n for which the higher-order construct Y is being created. In HAL, Y is required to be new. Also, it is often too difficult or even impossible to know whether Y will be called or not and, if so, at which program point. Thus, HAL follows a conservative approach while performing mode analysis and requires that the instantiation of the "captured" arguments (i.e., $X_1, \ldots, X_n$) remain unchanged after executing the predicate.

The above requirements allow us to follow a simple (although conservative) approach: Only after a call to Y will the trailing state of the captured variables be affected. If the predicate has many solutions (`multi` or `nondet`) and thus may involve backtracking, then the involved variables will be treated safely in the analysis at the call location.

If the predicate does not involve backtracking, then trailing information might not be inferred correctly at the call location if the call contains any unifications. This is because the captured variables are generally not known at the call location. To keep the trailing information safe any potential unifications have to be accounted for in the higher-order unification. Thus to be safe all captured variables have to be removed from the description.

Formally, let $x = \{X_1, \ldots, X_n\}$ and l_1 be the pre-description, then the post-description l_2 is:

$$l_2 = consist_2(l_1 \setminus x)$$

Higher-order call: $call(P, X_1, \ldots, X_n)$**.**
The exact impact of a higher-order call is impossible to determine in general. Fortunately, even if the exact predicate associated to variable P is unknown, the HAL compiler still knows its determinism. This can help us improve accuracy. If the determinism is `multi` or `nondet`, then all variables have to be trailed.

For any other determinism, all additional arguments $X_1, \ldots, X_n$ should be trailed afterwards, as they might have been involved in unifications. Also *consist* has to be called to bring any sharing with other variables into account.

Formally the the post-description l_2 is computed from the pre-description l_1 as follows:

$$l_2 = consist_2(l_1 \setminus \{X_1, \ldots, X_n\})$$

3.4 Optimisation based on the analysis

The pre-description of every unification is used to improve it:

- In variable–nonvariable unifications a variant of the unification without chain trailing can be used if the variable is in the pre-description.

- In variable–variable unifications a variant without swap trailing can be used if both involved variables are in the pre-description.

- In term construction no swap trailing is required when a variable is put in a structure if the variable is in the pre-description.

For some variables it cannot be statically determined whether they are free or bound. So HAL has a general unification predicate for each Herbrand type that perform boundness tests to find out at runtime whether to call a variable–variable, variable–nonvariable or nonvariable–nonvariable unification. The optimisation of these general kinds of unification with a trailing analysis pre-description consists of replacing the calls to the appropriate unifications with optimised ones, if appropriate. This means e.g. that for a general unification $X = Y$ where only X appears in the pre-description only the call to the variable–nonvariable unification, where X is the variable, can be replaced by an optimised one. If at runtime this case never occurs, then nothing has been gained by this optimisation.

4. RESULTS

4.1 Improved trailing scheme in dProlog

Here we present the experimental results of the improved conditional PARMA trailing scheme in dProlog for several smaller benchmarks and one bigger program, comp. Table 1 shows the timing and maximal trail use for each benchmark. The measurements were made on a 1.50 GHz Intel Pentium 4 with 256 MB of memory. Time is given in seconds and applies to the number of runs given in brackets. The maximal trail size is given in trail stack slots (words) and applies to a single run.

Benchmark	Time		Maximal trail	
	cparma	iparma	cparma	iparma
boyer(10)	.950	.920	115,366	57,683
browse(10)	1.010	1.010	2,326	1,163
cal(100)	1.800	1.800	112	56
chat(50)	1.020	1.040	932	498
crypt(2,000)	1.160	1.170	118	59
ham(20)	1.160	1.130	196	98
meta_qsort(1,250)	1.070	1.090	3,238	1,894
nrev(50,000)	.900	.860	112	56
poly_10(100)	.630	.650	13,470	6,735
queens_16(20)	1.810	1.790	176	88
queens(100)	3.310	3.300	170	85
reducer(200)	.440	.430	4,850	2,555
sdda(12,000)	1.000	1.010	326	201
send(100)	.800	.800	122	61
tak(100)	1.620	1.520	95,522	47,761
zebra(300)	2.510	2.730	406	205
relative average	100%	99.9%	100%	51.7%
comp(1)	1.930	1.890	644,170	337,856
comp relative	100%	97.9%	100%	52.4%

Table 1: Timing and maximal trail for the classic (cparma) and improved (iparma) conditional PARMA trailing scheme for dProlog.

The time difference between the classic and the improved scheme is negligible. The improved scheme is at most 8.8% slower, for the zebra benchmark, but on average both are about equally fast. The price for the lower trail usage is an increase in instructions executed and that is why there is no net speedup.

The differences in maximal trail use however are substantial. While swap trail and chain trail halve the trail stack consumption, value trailing is still used for some cases of variable–variable trailing. Yet experimental results show that that kind of variable–variable trailing does not occur very often in most benchmarks, as the maximal trail stack is effectively halved in eleven benchmarks and on average the maximal trail use is 51.7% of the classical scheme.

The results for the smaller benchmarks are confirmed by the larger comp program. Execution time is nearly the same for the classic and improved trailing scheme and the maximal trail shows a similar improvement of almost 50%.

4.2 Improved trailing scheme in HAL-Mercury

The improved unconditional PARMA trailing scheme has also been implemented in the HAL-Mercury system. Aside from the discussed trailings for unification this system also requires trailing when a term is constructed with an old variable as an argument. In this term construction the argument cell in the term structure is inserted in the variable chain. This modifies one cell in the old variable chain. In the classic scheme this cell is trailed with value trailing. To avoid value trailing altogether this has been replaced with swap trailing in the improved trailing scheme.

To compare the different trailing settings for the HAL-Mercury six benchmarks have been used. All timing results are in seconds and have been obtained on the same 1.50 GHz Intel Pentium 4 used for the benchmarks in the dProlog system. Execution times apply to the number of runs of the benchmarks recorded in table 2.

Benchmark	Runs
icomp	12,500
hanoi_difflist	2,500
qsort_difflist	25,000
serialize	12,500
warplan	10
zebra	200

Table 2: Number of runs the timing results of table 3, 4 and 5 apply to.

Table 3 presents the timing and maximal trail for both the classic and improved trailing scheme.

Benchmark	Time		Maximal trail	
	cparma	iparma	cparma	iparma
icomp	1.030	1.028	744	444
hanoi_difflist	1.001	.898	24,560	16,372
qsort_difflist	1.024	.969	1,208	804
serialize	1.065	1.034	1,696	1,048
warplan	1.587	1.624	1,816	1,256
zebra	1.030	1.121	1,064	532
relative average	100%	98.6%	100%	62.3%

Table 3: Timing and maximal trail for the classic (cparma) and improved (iparma) unconditional PARMA trailing scheme for HAL-Mercury.

For four out of six benchmarks the timing results favour the improved trailing scheme, while in two benchmarks the classic scheme is faster. The differences are a few percentages though, with a maximum difference of about 10% for the hanoi benchmark. Much more important are the effects of the improved trailing scheme on the maximal trail size. The maximal trail is at least 30% and up to 50% smaller for the improved scheme than for the classic scheme, but as in the dProlog case, the overhead of the extra instructions executed is sometimes larger than the gain.

4.3 Improved trailing scheme combined with trailing analysis in HAL-Mercury

The trailing analysis presented in Section 3 was implemented in the HAL analysis framework in order to detect unnecessary trailings for the improved trailing scheme. Table 4 presents the timing and maximal trail for the benchmarks under the improved scheme combined with analysis and compares the results with the same scheme without analysis.

Benchmark	Time		Maximal trail	
	iparma	iaparma	iparma	iaparma
icomp	1.028	.959	444	404
hanoi_difflist	.898	.734	16,372	0
qsort_difflist	.969	.765	804	0
serialize	1.034	1.034	1,048	1,048
warplan	1.624	1.624	1,256	1,256
zebra	1.121	1.095	532	476
relative average	100%	91.9%	100%	63.4%

Table 4: Timing and maximal trail for the improved unconditional PARMA scheme with (iaparma) and without (iparma) trailing analysis.

For the serialize and warplan benchmarks the analysis was not able to reduce the number of actual trailing operations. For the other four benchmarks the combination of the improved scheme with analysis yields better results, both for time and maximal trail. For the hanoi and qsort benchmarks there is a drastic improvement: all trailings have been avoided, with a distinctive time improvement of about 20%. For the other two benchmarks, icomp and zebra, there is a maximal trail improvement of about 10% together with a slightly reduced time, 7% and 3% better respectively. Overall the combination of the improved scheme with the trailing analysis never makes the results worse. Since it drastically improves some benchmarks and shows a modest improvement of others, it is fair to conclude that the combination is superior to the improved system without analysis.

4.4 Comparison of the classic and improved scheme with respective trailing analyses

[13] proposes a trailing analysis on top of the classic PARMA trailing scheme. Table 5 compares the current work with the results obtained in [13].

The timing differences between the two systems with analysis show a similar pattern as those without analysis: for some benchmarks the improved systems is better because more trailings were avoided, for others it is worse because of additional instruction overhead.

The maximal stack trail of the improved system with analysis is smaller for four benchmarks: from 20% to 50%. For the hanoi and qsort benchmarks the maximal stack trail obtained is the same: all trailings are avoided. The remarkable time difference for the qsort benchmark then is not due to a different number of trailings but must be caused by differences in the branches of the general unification predicates (see Section 3.4) that are not actually executed.

Benchmark	Time		Maximal trail	
	caparma	iaparma	caparma	iaparma
icomp	.991	.959	608	404
hanoi_difflist	.739	.734	0	0
qsort_difflist	.803	.765	0	0
serialize	.999	1.034	1,296	1,048
warplan	1.580	1.624	1,816	1,256
zebra	1.000	1.095	952	476
relative average	100%	101.2%	100%	77.7%

Table 5: Timing and maximal trail for the classic (caparma) and improved (iaparma) unconditional PARMA trailing scheme for HAL-Mercury, both with trailing analysis.

5. RELATED AND FUTURE WORK

The improvement to the classic PARMA trailing scheme is new as far as we know.

A somewhat similar analysis for detecting variables that do not have to be trailed is presented by Debray in [2] together with corresponding optimisations. Debray's analysis however is for the WAM variable representation and in a traditional Prolog setting, i.e., without type, mode and determinism declarations. Also in [21] trailing is avoided, but only for variables that are *new* in our terminology and again, the setting is basically the WAM representation.

Taylor too keeps track of a trailing state of variables in the global analysis of his PARMA system with the classic PARMA trailing scheme (see [18, 17]). As opposed to the analysis we have presented in [13] for HAL, Taylor's analysis is less precise and more similar to the analysis we have presented here: the trailing state of a variable can only be that is has to be trailed or not. In [13] there is an intermediate trailing state, shallow trailed, meaning that the associated cell of a variable, the one directly pointed to, does not have to be trailed.

There exists also a runtime technique for preventing the multiple value trailing between two choice points: see for instance [11]. However, this technique only works in the WAM scheme, because it introduces linear reference chains that PARMA does not allow.

Finally, [14] and [20] also discuss the reconstruction of a state on backtracking by copying and recomputing techniques respectively. The context of those works is quite different, but the latter technique is naturally related to a hybrid technique mentioned later in this section.

There is little room left for optimisation of the trailing analysis for the improved unconditional trailing scheme. Of course the analysis itself can be improved by adopting a more refined representation for bound variables. Now all PARMA chains in the structure of a bound variable are represented by the same trailing state. Bound variables could be represented more accurately, by requiring the domain to

keep track of the different chains contained in the structures to which the program variables are bound, their individual trailing state and how these are affected by the different program constructs. Known techniques (see for instance [7, 6, 9]) based on type information could be used to keep track of the constructor that a variable is bound to and the trailing state of the different arguments, thereby making this approach possible. This applies equally to the analysis of the classical scheme.

Additionally it would be interesting to see how much extra gain analysis can add to the improved conditional trailing scheme as implemented in dProlog or in a HAL compilation scheme to Mercury grade that supports conditional trailing. Such analysis would certainly not improve the maximal trail, but it would remove the overhead of the runtime test. This will most likely also result in a small speed-up.

Though experimental results show that the improved scheme with analysis is better than the classic scheme with analysis, this need not be true for all programs. Recall that between two choice points all value trailings of a cell but the first can be eliminated in the classic scheme, while no swap trailings could be eliminated in the improved scheme. A hybrid scheme with an analysis that decides on a single unification basis if either swap trailing or value trailing is better at minimising the amount of trailing and the cost of untrailing. This analysis would require a more global view of all the trailings in between two choice points. Moreover some trailings could be common to different pairs of choice points and optimality would depend on where execution spends most of its time.

Also the untrailing operation can be improved: when analysis is able to determine for instance that the only trailing that happened was a swap trailing, no tags need to be set and tested.

6. REFERENCES

[1] H. Aït-Kaci. *Warren's Abstract Machine: A Tutorial Reconstruction*. MIT Press, 1991.

[2] S. Debray. A Simple Code Improvement Scheme for Prolog. *JLP*, 13(1):349–366, May 1992.

[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. Moniz 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, M. García de la Banda, W. Harvey, K. Marriott, and P. J. Stuckey. Herbrand Constraint Solving in HAL. In *International Conference on Logic Programming*, pages 260–274, 1999.

[5] B. Demoen, M. García de la Banda, W. Harvey, K. Marriott, and P. J. Stuckey. An Overview of HAL. In *Principles and Practice of Constraint Programming*, pages 174–188, 1999.

[6] P. V. Hentenryck, A. Cortesi, and B. L. Charlier. Type analysis of Prolog using type graphs. *JLP*, 22:179–209, 1995.

[7] G. Janssens and M. Bruynooghe. Deriving descriptions of possible value of program variables by means of abstract interpretation. *JLP*, 13:205–258, 1993.

[8] T. Lindgren, P. Mildner, and J. Bevemyr. On Taylor's scheme for unbound variables. Technical report, Computer Science Department, Uppsala University, October 1995.

[9] A. Mulkers, W. Winsborough, and M. Bruynooghe. Live-structure dataflow analysis for prolog. *ACM TOPLAS*, 16:205–258!, 1994.

[10] N. Nethercote. The Analysis Framework of HAL. Master's thesis, University of Melbourne, September 2001.

[11] J. Noyé. *Elagage de contexte, retour arrière superficiel, modifications réversibles et autres: une étude approfondie de la WAM*. PhD thesis, Université de Rennes I, Nov. 1994.

[12] M. García de la Banda, K. Marriott, P. Stuckey, and H. Søndergaard. Differential methods in logic program analysis. *JLP*, 35:1–37, 1998.

[13] T. Schrijvers, M. García de la Banda, and B. Demoen. Trailing Analysis for HAL. In P. J. Stuckey, editor, *Proceedings of the Eighteenth International Conference on Logic Programming*, pages 37–51, Copenhagen, Denmark, July 29–Aug. 1 2002. Springer Verlag.

[14] C. Schulte. Comparing trailing and copying for constraint programming. In D. De Schreye, editor, *Proceedings of the Sixteenth International Conference on Logic Programming*, pages 275–289, Las Cruces, NM, USA, Nov. 1999. The MIT Press.

[15] S. Shapiro. *The Art of Prolog*. The MIT Press, 1996.

[16] Z. Somogyi, F. Henderson, and T. Conway. Mercury: an efficient purely declarative logic programming language. In *Australian Computer Science Conference*, pages 499–512, February 1995.

[17] A. Taylor. Removal of Dereferencing and Trailing in Prolog Compilation. In *Sixth Internation Conference on Logic Programming*, pages 48–60. MIT Press, 1989.

[18] A. Taylor. *High Performace Prolog Implementation*. PhD thesis, Basser Department of Computer Science, June 1991.

[19] A. Taylor. Parma - Bridging the Performance GAP Between Imperative and Logic Programming. *JLP*, 29(1-3):5–16, 1996.

[20] P. Van Hentenryck and V. Ramachandran. Backtracking without Trailing in CLP(Rlin). *ACM Transactions on Programming Languages and Systems*, 17(4):635–671, July 1995.

[21] P. Van Roy and A. M. Despain. High-Performance Logic Programming with the Aquarius Prolog Compiler. *IEEE Computer*, 25(1):54–68, 1992.

[22] D. Warren. An Abstract Prolog Instruction Set. Technical Report 309, SRI International, Menlo Park, U.S.A., October 1983.