Trailing Analysis for HAL

Tom Schrijvers[1], Maria García de la Banda[2] and Bart Demoen[1]

[1] Dept. of Computer Science, K.U.Leuven, Belgium
[2] Dept. of Computer Science, Monash University, Melbourne

Abstract. The HAL language includes a Herbrand constraint solver which uses Taylor's PARMA scheme rather than the standard WAM representation. This allows HAL to generate more efficient Mercury code. Unfortunately, PARMA's variable representation requires value trailing with a trail stack consumption about twice as large as for the WAM. We present a trailing analysis aimed at determining which Herbrand variables do not need to be trailed. The accuracy of the analysis comes from HAL's semi-optional determinism and mode declarations. The analysis has been partially integrated in the HAL compiler and benchmark programs show good speed-up.

1 Introduction

Mercury [SHC95] is a logic programming language considerably faster than traditional Prolog implementations. One reason is that Mercury requires the programmer to provide type, mode and determinism declarations whose information is used to generate efficient target code. Another reason is that variables can only be ground (i.e., bound to a ground term) or free (i.e., first time seen by the compiler and thus unbound and unaliased). Since neither aliased variables nor partially instantiated structures are allowed, Mercury does not need to support full unification; only assignment, construction, deconstruction and equality testing for ground terms are required. Furthermore, it does not need to perform trailing. This is because trailing aims at storing enough information to be able to reconstruct the previous state upon backtracking. This usually means recording the state of unbound variables right before they become aliased or bound. Since free variables have no runtime representation they do not need to be trailed.

HAL [DdlBH+99b,DdlBH+99a] is a constraint logic language designed to support the construction, extension and use of constraint solvers. HAL also requires type, mode and determinism declarations and compiles to Mercury 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 [Tay91,Tay96] rather than the standard WAM representation [AK91], because, unlike the WAM, the PARMA representation for ground terms is equivalent to that of Mercury. Thus, calls to the Herbrand constraint solver can be replaced by calls to Mercury's efficient routines whenever ground terms are being manipulated.

Unfortunately, the increased expressive power of full unification comes at a cost, which includes the need to perform trailing. Furthermore, trailing is more expensive in the PARMA scheme than in the WAM. This overhead can however be reduced by performing a trailing analysis that detects and eliminates unnecessary trailings. For traditional logic languages such analysis is rather inaccurate, since little is known about the way predicates are used. For HAL however, determinism information significantly improves accuracy, thus countering the negative aspects of the PARMA scheme and helping retain Mercury-like efficiency.

Next section reviews Taylor's scheme, the trailing of PARMA variables and when it can be avoided. Section 3 summarises the information used by our domain to improve its accuracy. Section 4 presents the `notrail` analysis domain. Section 5 shows how to analyse HAL's body constructs. Section 6 shows how to use the analysis information to avoid trailing. The results of the analysis are summarised in Section 7. Finally, future work is discussed in Section 8.

2 The PARMA Scheme and Trailing

An unbound variable is represented in the PARMA scheme by what is known as a PARMA chain. If the variable is not aliased the chain has length one (a self-reference). For a set of aliased variables the chain is a circularly linked list. Unifying two variables in this scheme consists of cutting their PARMA chains and combining them into one big chain. When a variable becomes bound, all cells in its chain are set to reference the structure that it is bound to. This is unlike the WAM scheme, where only one cell (the one obtained by dereferencing) is set to reference the structure. Hence, checking whether a variable is bound in the PARMA scheme requires no dereferencing, thus increasing efficiency.

As mentioned before, trailing aims at storing enough information regarding the representation state of a variable before each choice-point to be able to reconstruct such state upon backtracking. In the case of PARMA chains the change of representation state occurs at the cell level: from being a self-reference (when the variable pointing to the cell – *the associated variable* – is unbound and unaliased), to pointing to another cell in the chain (when the associated variable gets aliased), to pointing to the final structure (when any variable associated to a cell in the same chain gets bound). Thus what we need to trail are the cells. This is done in HAL using the following code:

```
trail(p, tr)
{
   *(tr++) = *p;
   *(tr++) = p;
}
```

which takes a pointer `p` to a cell in a PARMA chain and the pointer `tr` to the top of the trail. It first stores the contents of the cell and then its address.

Let us now discuss when cells need to be trailed and when this can be avoided. We have seen before that trailing is only needed when the representation state of a variable changes, and that this can only happen when the variable is unbound and, due to a unification, it becomes either aliased or bound.

Trailing during variable–variable unification Let us start by discussing the information needed to reconstruct the state before aliasing two unbound variables belonging to separate chains. The result of the aliasing is the merging of the two separate chains into a single one. This can be done by changing the state of only two cells: those associated to each of the variables. Since each associated cell appears in a different chain, the final chain can be formed by simply interchanging their respective successors. To be able to reconstruct the previous situation, one just needs to know which two cells have been changed and what their initial value was. This is achieved by the following (simplified) code:

```
trail(X,tr);
trail(Y,tr);
oldX = *X;
X = *Y;
Y = oldX;
```

Notice that `X` and `Y` are trailed independently. As only their associated cells need to be trailed, we will refer to this kind of trailing as *shallow* trailing.

Trailing during variable–nonvariable unification When an unbound variable is bound every single cell in its chain is set to point to the nonvariable term. Thus, we can only reconstruct the chain if *all* cells in the chain are trailed. Note that this is much more expensive than in the WAM scheme where only the dereferenced cell is set to point to the nonvariable term and, therefore, only this cell needs to be trailed. The combined unification-trailing code is as follows:

```
start = X;
do {
   next = *X;
   trail(X,tr);
   *X = T;
   X = next;
} while (X != start);
```

Since all cells in the chain of the unbound variable are trailed, we call this *deep* trailing of the variable.

Unnecessary trailing: There are at least two cases in which the trailing of an unbound variable can be avoided:

- The variable had no representation before the unification (e.g., it was free in Mercury): there is no previous value to remember, so trailing is not required.
- The cells that need to be trailed (the associated cell in the case of variable–variable, all cells in the case of variable–nonvariable) have already been trailed *since the most recent choice-point*. Upon backtracking only the earliest trailing after the choice-point is important, since that is the one which enables the reconstruction of the state of the variable before the choice-point.

3 Language Requirements

The analysis presented in this paper was designed for the HAL language. However, it can be useful for any language that uses PARMA representation and that provides accurate information regarding the following properties:

- Instantiation state: trailing analysis can gain accuracy by taking into account the instantiation state, i.e. whether the variable is new, ground or old. State new corresponds to variables with no internal representation (equivalent to Mercury's free instantiation). State ground corresponds to variables known to be bound to ground terms. In any other case the state is old, corresponding to variables which might be unbound but do have a representation (a chain of length one or more) or bound to a term not known to be ground. Variables with instantiation 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 it is known to be ground, it can safely remain ground. Thus, the three states can be considered mutually exclusive. The information should be available at each program point p as a table associating with each variable in scope of p its instantiation state.
- Determinism: trailing analysis can also gain accuracy from the knowledge that particular predicates have at most one solution. This information should be available as a table associating with each predicate (procedure to be more precise) its inferred determinism.
- Sharing: our trailing analysis can exploit sharing information to increase accuracy. This information should be available at each program point p as a table associating with each variable in scope of p the set of variables which possibly share with it.

4 The `notrail` Analysis Domain

The aim of the `notrail` domain is to keep enough information to be able to decide whether the program variables in a unification need to be trailed or not, so that if possible, optimised versions which do not perform the trailing can be used instead. In order to do this, we must remember that only variables which are unbound at run-time need to be trailed. This suggests making use of the instantiation information mentioned in the previous section.

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. Assuming that Var_p contains n variables and the tree we have used to implement the underlying table is sufficiently balanced, then, the size of the Old_p is $\mathcal{O}(n)$ and the complexity of $inst_p$ is $\mathcal{O}(\log n)$.

We have already established that variables in New_p and $Ground_p$ do not need to be trailed. Thus, only variables in Old_p need to be represented in the `notrail` domain. Recall that Old_p not only contains all unbound program variables, but also those bound to terms which the analysis cannot ensure to be ground. This is necessary to ensure correctness: even though variables which are bound do not need to be trailed, program variables might be bound to terms containing one or more unbound variables. It is the trailing state of these unbound run-time variables that is represented through the domain representation of the bound program variable.

Now that we have decided which program variables need to be represented by our domain, we have to decide how to represent them. We saw before that it is unnecessary to trail a variable in a variable–variable unification if its associated cell has already been trailed, i.e., if the variable has already been shallow trailed since the most recent choice-point. For the case of variable–nonvariable unification this is not enough, we need to ensure all cells in the chain have already been trailed, i.e, the variable has already been deep trailed. This suggests a domain which distinguishes between shallow and deep trailed variables. This can be easily done by partitioning Old_p into three disjoint sets of variables with a different trailing state: those which might not have been trailed yet, those which have at least been shallow trailed, and those which have been deep trailed. It is sufficient to represent only two sets to be able to reconstruct the third. Hence, the type of the elements of our `notrail` domain $L_{\texttt{notrail}}$ will be $\mathcal{P}(Old_p) \times \mathcal{P}(Old_p)$, where the first component represents the set of variables which have already been shallow trailed, and the second component represents the set of already deep trailed variables. In the following we will use $l_1, l_2, \ldots$ to denote elements of $L_{\texttt{notrail}}$ at program points $1, 2, \ldots$, and $s_1, s_2, \ldots$ and $dp_1, dp_2, \ldots$ for the already shallow and deep trailed components of the corresponding elements. Also, the elements of the domain will be referred to as descriptions. The descriptions before and after a goal will be referred to as the pre- and post-descriptions, respectively.

Note that, by definition, we can state that if a variable has already been deep trailed, then it has also been shallow trailed (i.e., if all cells in the chain have already been trailed, then the cell associated to the variable has also been trailed). The partial ordering relation $\sqsubseteq$ on $L_{\texttt{notrail}}$ is thus defined as follows:

$$\forall (s_p^1, dp_p^1), (s_p^2, dp_p^2) \in L_{notrail} : (s_p^1, dp_p^1) \sqsubseteq (s_p^2, dp_p^2) \Leftrightarrow \begin{cases} s_p^2 \subseteq dp_p^1 \cup s_p^1 \\ dp_p^2 \subseteq dp_p^1 \end{cases}$$

This implies that deep trailing is stronger information than shallow trailing, and shallow trailing is stronger than no trailing at all. Also note that descriptions are compared at the same program point only (so that the instantiation and sharing information is identical). An example of a trailing lattice is shown in Fig. 1. Clearly $(L_{notrail}, \sqsubseteq)$ is a complete lattice with top description $\top_p = (\emptyset, \emptyset)$ and bottom description $\bot_p = (\emptyset, Old_p)$.

There are two important points that need to be taken into account when considering the above domain. The first point is that the dp_p component of a description will be used not only to represent already deep trailed variables but any variable in Old_p which for whatever reason does not need to be trailed. The reader might then wonder why variables in New_p or $Ground_p$ are not included

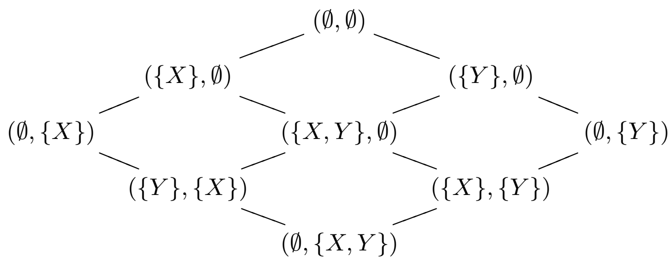


Fig. 1. `Notrail` lattice example

in dp_p since they do not need to be trailed either. This is of course possible. However, this would make most abstract operations slightly more complex.

The second point is that as soon as a deep trailed variable X is known to share with a shallow trailed variable Y, X also must become shallow trailed since some cell in some newly merged chain might come from Y and thus might not have been trailed. We will thus use the sharing information provided at each program point p to define the function $share_p : Old_p \to \mathcal{P}(Old_p)$, 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:

$$consist_p((s, dp)) = (s \cup x, dp \setminus x)$$

where

$$x = \{X \in dp | (share_p(X) \setminus dp) \neq \emptyset\}$$

From now on we will assume that $\forall(s, dp) \in L_{\texttt{notrail}} : consist_p((s, dp)) = (s, dp)$ and use the *consist* function to preserve this property.

Given HAL's implementation of the sharing analysis domain `ASub` [Søn86] the time complexity of $share_p$ is $\mathcal{O}(n^2)$. Furthermore, since `ASub` explicitly carries the set of ground variables at each program point (g_p), we will use this set rather than computing a new one ($Ground_p$) from the instantiation information, thus increasing efficiency. The major cost of $consist_p$ is the computation of x: for each of the $\mathcal{O}(n)$ variables the $share_p$ set has to be computed. All other set operations are negligible in comparison. Hence, the overall time complexity is $\mathcal{O}(n^3)$. We will see that the complexity of this function determines the complexity of all the operations that use it. Thus, we will use it only when strictly necessary.

In summary, each element $l_p = (s_p, dp_p)$ in our domain can be interpreted as follows. Consider the program variable X. If $X \in dp_p$, this means that all cells in all chains represented by X have already been trailed (if needed). Therefore, X does not need to be trailed in any unification for which l_p is a pre-condition. If $X \in s_p$ we have two possibilities. If X is known to be unbound, then its associated cell has been shallow trailed. Therefore, it does not need to be trailed in any variable-variable unification for which l_p is a pre-condition. If X might be bound, then a cell of one of its chains might not be trailed. As a result, no optimisation can be performed in this case.

We could, of course, represent bound variables 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 [JB93,HCC95,MWB94]) based on type information could be used to keep track of the constructor that a variable is bound to and of the trailing state of the different arguments, thereby making this approach possible.

5 Analysing HAL Body Constructs

This section defines the `notrail` operations required by HAL's analysis framework [Net01] to analyse the different body constructs.

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

- If one of the variables (say X) is new, it will simply be assigned a copy of the tagged pointer of Y. No new PARMA chain is created, and thus no trailing is required. The trailing state of X becomes the same as that of Y.
- If one of the variables is ground, the other one will be ground after the unification. Hence, neither of them will appear in the post-description.
- If both variables are deep trailed, the unification only needs to merge the chains (no need to trail again). Hence, both variables remain deep trailed.
- Otherwise, at least one of the variables is not deep trailed. If both variables are unbound, unification will merge both chains while at the same time performing shallow trailing if necessary. Thus after the unification both variables will be shallow trailed. If at least one variable is bound, the other one will become bound after the unification. As stated earlier, bound variables can be treated in the same way.
  Note that if either variable was deep trailed before the unification, all shared variables must become shallow trailed as well after the unification. This requires applying the *consist* function.

Formally, let $l_1 = (s_1, dp_1)$ 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:

$$l_2 = unify(X,Y) = \begin{cases} same(X,Y,l_1) & , X \textit{ is new} \\ remove\_ground(l_1, g_2) & , X \textit{ is ground} \\ min(X,Y,l_1) & , X \textit{ and } Y \textit{ are old} \\ unify(Y,X) & , \textit{otherwise} \end{cases}$$

with

$$remove\_ground(l_i, v_i) = (s_i \setminus v_i, dp_i \setminus v_i)$$

$$same(X,Y,(s_1,dp_1)) = \begin{cases} (s_1 \cup \{X\}, dp_1) & , Y \in s_1 \\ (s_1, dp_1 \cup \{X\}) & , Y \in dp_1 \\ (s_1, dp_1) & , \textit{otherwise} \end{cases}$$

$$min(X,Y,(s_1,dp_1)) = \begin{cases} (s_1, dp_1) & , \{X,Y\} \subseteq dp_1 \\ consist_2((s_1 \cup \{X,Y\}, dp_1 \setminus \{X,Y\})) & , \textit{otherwise} \end{cases}$$

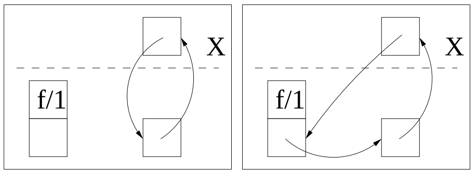
Here $remove\_ground(l_i, v_i)$ removes all variables in v_i from l_i, $same(X, Y, l_i)$ gives X the same trailing state as Y and $min(X, Y, l_i)$ ensures that X and Y have a shallow trailed state, unless both are deep trailed.

Note that the first case in the definition of *same* does not require a call to *consist* even though dp_1 is modified by adding X to it. This is because X was previously a new variable and, thus, it cannot introduce any sharing.

The worst case time complexity, $\mathcal{O}(n^3)$, is again due to *consist*.

Variable–term unification: $Y = f(X_1, \ldots, X_n)$. There are two cases to consider: If Y is new, 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.

When a term, e.g. $f(X)$, is constructed with X being represented by a PARMA chain, the argument cell in the structure representation of $f/1$ is inserted in the chain of X (see Fig.2). While X requires shallow trailing, the cell of the term requires no trailing at all as it is newly created.



(a) Before. (b) After.

Fig. 2. Term construction example: $f(X)$. The dashed line represents a choice-point.

Generalised, as the arguments of a new term will share after term construction, they are only allowed to remain deep trailed if all are deep trailed. Otherwise all arguments become shallow trailed. Similarly, Y becomes deep trailed if all arguments are deep trailed, and shallow trailed otherwise.

Formally, let $l_1 = (s_1, dp_1)$ 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:

$$l_2 = \begin{cases} (s_1, dp_1 \cup \{Y\}) & , x \subseteq dp_1 \\ (s_1 \cup x \cup \{Y\}, dp_1) & , x \cap dp_1 = \emptyset \\ consist_2((s_1 \cup x \cup \{Y\}, dp_1 \setminus x)) & , otherwise \end{cases}$$

The worst case time complexity is $\mathcal{O}(n^3)$. This definition can be combined with the previous one for the overall definition of variable–term unification. The implementation can be more efficient, but the complexity will still be $\mathcal{O}(n^3)$.

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:

$$onto\_proj(l, v) = (s \cap v, dp \cap v)$$

The second step consists in 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 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 determinism information. 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 predicate might have more than one answer, then l_2 is equal to l_{result} 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, become not 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. Any deep trailed variables that share with non-deep trailed variables must, of course, become shallow trailed.

Formalised, the combination[3] function is defined as:

$$\begin{aligned} l_2 &= comb(l_1, l_{answer}) \\ &= \begin{cases} consist(((s_1 \setminus x) \cup s_{answer}, (dp_1 \setminus x) \cup dp_{answer})) & \text{, at most one solution} \\ consist(l_{answer}) & \text{, otherwise} \end{cases} \end{aligned}$$

Obviously the complexity is $\mathcal{O}(n^3)$ because of *consist*.

Example 1. Assume that the call `p(X)` has pre-description $(\{X, Y\}, \emptyset)$ and the predicate `p/1` has answer description $(\{X\}, \emptyset)$. The post-description of the call depends on the determinism of the predicate. If the predicate has at most one solution, the post-description will be $((\{X, Y\} \setminus \{X\}) \cup \{X\}, (\emptyset \setminus \{X\}) \cup \emptyset) = (\{X, Y\}, \emptyset)$. Otherwise the post-description will be equal to the answer description, $(\{X\}, \emptyset)$.

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

[3] Note that the combination is not the meet of the two descriptions. It is the "specialised combination" introduced in [dlBMSS98] 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.

with a call description equal to l_{proj}. If not, then we will choose the smallest call-description less precise than l_{proj}. Since the table *always* includes a pair with the most general ($\top$) calling description, this selection process always 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. As mentioned before, 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 $l_i = (s_i, dp_i), 1 \leq i \leq n$ 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, where out-projection is identical to *remove_ground*, which has time complexity $\mathcal{O}(n)$. The end result l_{n+1} of the disjunction is the least upper bound (lub) of all branches[4], which is defined as:

$$l_1 \sqcup \ldots \sqcup l_n = consist_{n+1}(remove\_ground((s, dp), g_{n+1}))$$

where
$$\begin{aligned} s &= (s'_1 \cap \ldots \cap s'_n) \setminus dp \\ dp &= (dp'_1 \cap \ldots \cap dp'_n) \\ s'_i &= s_i \cup dp'_i \\ dp'_i &= dp_i \cup g_i \end{aligned}$$

Intuitively, all variables which are deep trailed in all descriptions are ensured to remain deep trailed; all variables which are trailed in all descriptions but have not always been deep trailed (i.e., are not in dp) are ensured to have already been (at least) shallow trailed. Note that variables which are known to be ground in all descriptions (those in g_{n+1}) are eliminated. This is consistent with the view that only old variables are represented by the descriptions.

Example 2. Let $l_0 = (\emptyset, \{X, Y, Z\})$ be the pre-description of disjunction:

```
( X = Y ; X = f(Y, Z) )
```

Let us assume there is no sharing at that program point. Then, the pre-descriptions of the first unification is $(\emptyset, \emptyset)$, the $\top$ element of our domain. The pre-description of the second unification is $(\emptyset, \{X, Y, Z\})$, i.e., since this is the last branch in the disjunction, its pre-description is identical to the pre-description of the entire disjunction. Their post-descriptions are $(\{X, Y\}, \emptyset)$ and $(\emptyset, \{X, Y, Z\})$, respectively. Finally, the lub of the two post-descriptions results in $(\{X, Y\}, \emptyset)$.

The time complexity of the joining of the branches is simply that of the lub operator ($\mathcal{O}(n^3)$) for a fixed maximum number of branches, and it is completely dominated by the $consist_{n+1}$ function.

---

[4] Note that this is not the lub of the `notrail` domain alone, but that of the product domain which includes sharing (and groundness) information.

If-then-else: $I \;\rightarrow\; T \,;E$. Although the if-then-else could be treated as $(I, T; E)$, this is rather inaccurate since only one branch will ever be executed and, thus, there is no backtracking between the two branches.

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. The time complexity of the joining of the branches is again $\mathcal{O}(n^3)$, just like the operation over the disjunction.

Example 3. Let $l_0 = (\emptyset, \emptyset)$ be the pre-description of the if-then-else:

```
( N = 1 →
        X = Y
;
        X = f(Y, Z)
)
```

Assume no variables share before the if-then-else. Then, l_0 is equal to the pre-description of both the then- and else-branch. The post-description of the then-branch is $(\{X, Y\}, \emptyset)$ and that of the else-branch is $(\{X, Y, Z\}, \emptyset)$. The post-description finally is obtained by taking their lub: $(\{X, Y\}, \emptyset)$.

Higher-order unification: $Y = p(X_1, \ldots, X_n)$. 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, where. Thus, HAL follows a conservative approach 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 of the captured variables be affected. If the predicate might have more than one solution and thus may involve backtracking, then the involved variables will be treated safely in the analysis at the call location if they are still statically live there.

If the predicate does not involve backtracking, then trailing information might not be inferred correctly at the call location if the call contains 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. Since the predicate involves no backtracking and all unifications leave the variables they involve at least shallow trailed, it is sufficient to demote all captured deep trailed variables to shallow trailed status, together with all sharing deep trailed variables.

Formally, let $l_1 = (s_1, dp_1)$ be the pre-description of the higher-order unification and x be the set of variables $\{X_1, \ldots, X_n\}$. Then its post-description l_2 can be obtained with a time complexity of $\mathcal{O}(n^3)$ as:

$$l_2 = \begin{cases} consist_2((s_1 \cup (x \cap dp_1), dp_1 \setminus x)) & , x \cap dp_1 \neq \emptyset \\ l_1 & , otherwise \end{cases}$$

Notice that between the time of higher-order unification and call, any number of disjunctions could occur. This means that the trailing state of the captured variables at the time of higher-order unification cannot generally be used to select another variant of the predicate than the one with top calling description.

Higher-order call: call$(P, X_1, \ldots, X_n)$. The exact impact of a higher-order call is difficult 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 predicate might have more than one solution, all variables must become not trailed. Since the called predicated is typically unknown, no answer description is available to improve accuracy.

Otherwise, the worst that can happen is that the deep trailed arguments of the call become shallow trailed. So in the post-description we move all deep trailed arguments to the set of shallow trailed variables, together with all variables they share with. Recall that for this case the captured variables have already been taken care of at the higher-order unification.

The sequence of steps is much the same as that for the predicate call. First, we project the pre-description l_1 onto the set x of variables $\{X_1, \ldots, X_n\}$, resulting in l_{proj}. Next, the answer description l_{answer} of the higher-order is computed as indicated above:

$$l_{answer} = \begin{cases} (s \cup dp, \emptyset) & \text{, at most one solution} \\ (\emptyset, \emptyset) & \text{, otherwise} \end{cases}$$

Finally, the combination of l_{answer} and l_1 is computed to obtain the post-description l_2.

6 Trailing Optimisation

The optimisation phase consists in deciding for each unification in the body of a clause, which variables need to be trailed. This decision is based on the pre-description of the unification, inferred by the trailing analysis. If some variables do not need to be trailed, the general unification predicate is replaced with an alternative variant that does not trail those particular variables. Thus, we will need a different variant for each possible combination of variables that do and do not need to be trailed.

– For the unification of two unbound variables trailing is omitted for either variable if it is shallow trailed or deep trailed in the pre-description.

– For the binding of an unbound variable trailing is omitted if the variable is deep trailed in the pre-description.
– For the unification of two bound variables the trailing for chains in the structure of either is omitted if the variable is deep trailed in the pre-description.

Often it is not known at compile time whether a variable is bound or not, so a general variable-variable unification predicate is required that performs runtime boundness tests before selecting the appropriate kind of unification. Various optimised variants of this general predicate are needed as well.

Finally, we must point out that term construction with old unbound arguments also involves trailing. This is because each argument is copied into the term structure and if a copy appears to be a pointer to a PARMA chain (the argument was an old unbound variable), then the copy is transformed into a new cell that is added in front of the cell C associated with the argument variable X. Since cell C is modified in the process it requires trailing. However, if X is either shallow or deep trailed, then trailing can be omitted.

7 Results

The analysis has been implemented in the analysis framework of HAL and applied to six HAL benchmarks that use the Herbrand solver: icomp, hanoi_difflist, qsort_difflist, serialize, warplan and zebra. The pre-descriptions inferred for the unifications have then been used to optimise the generated Mercury code by the omission of trailing, as explained in the previous section.

Benchmark	Compilation Time			Old unifications	
	Analysis	Total	Percentage	Improved	Total
icomp	34.670	84.760	40.9%	300	1269
hanoi_difflist	.630	7.720	8.2%	13	13
qsort_difflist	.500	32.590	1.5%	7	7
serialize	1.950	8.120	24.0%	10	17
warplan	22.120	98.160	22.5%	69	1392
zebra	2.830	12.550	22.5%	41	178

Table 1. Compilation statistics

Table 1 shows compilation statistics of the benchmarks: compilation time in seconds, measured on an Intel Pentium 166 MHz 96 MB, and the number of improved unifications compared to the total number of unifications involving old variables. The compilation times are high for most benchmarks because most predicates have many call descriptions to consider, something the analysis has not been optimised for yet. For the hanoi_difflist and qsort_difflist benchmarks, the analysis infers that all unifications should be replaced by a non-trailing alternative. In the other benchmarks a much smaller fraction of unifications can be improved due to the heavy use of non-deterministic predicates.

Table 2 presents the timing results of each benchmark for the given number of times it is performed, and compares the number of value trailings of the

Benchmark	iterations	Time		Speed-up	Value trailings	
		unoptimised	optimised		unoptimised	optimised
icomp	12,500	1.033	.992	4.0%	146	121
hanoi_difflist	2,500	.973	.739	24.0%	190	0
qsort_difflist	25,000	1.005	.775	22.9%	151	0
serialize	12,500	1.083	1.004	7.3%	212	162
warplan	10	1.663	1.613	3.0%	10,229	10,229
zebra	200	1.124	1.049	6.7%	25,769	24,618

Table 2. Timings and trailings.

unoptimised and optimised versions of the benchmarks for a single run. Timing results were obtained on an Intel Pentium 4 1.50 GHz 256 MB.

The huge speed-up of hanoi_difflist and qsort_difflist can be explained by the complete elimination of value trailing. For the other benchmarks the number of eliminated trailings and the speed-up are a lot smaller. This is partly explained by the smaller fraction of improved unification predicates and partly because for some of the improved general unifications the variables are always bound and thus there is no trailing to avoid.

8 Related and Future Work

In [SD02] an improved PARMA trailing scheme is proposed that nearly halves the maximal trail stack for conditional trailing. Also in the HAL-Mercury setting a maximal trail stack reduction of 30% to 50% is obtained. The analysis has been adapted for this improved trailing scheme, and, although it yields less additional improvement, the combination with this trailing scheme is more effective.

A somewhat similar analysis for detecting variables that do not have to be trailed is presented by Debray in [Deb92] together with corresponding optimisations. Debray's analysis however is for the WAM variable representation in a traditional Prolog setting, i.e., without type, mode and determinism declarations. Our analysis could not be easily adapted to the WAM, as it relies heavily on the PARMA variable representation.

Since the presented analysis aims at avoiding trailing, it is worth comparing with runtime methods. If the underlying implementation has an ordered heap (like the WAM) the conditional trailing test is simple: the address of the cell is compared with the heap top of the most recent choice-point. In other cases, the conditional trailing test must rely on some form of time stamping. It is clear that runtime tests avoid **all** unnecessary trailing and analysis cannot improve this. Also, our benchmarks show that there is little to improve (see results for hanoi and qsort). Finally, the runtime tests can be combined with the analysis, but our feeling is that this should not be done with the expensive time stamping mechanism. Note that the technique of [Noy94] for avoiding double trailing of the same cell cannot be applied in the PARMA context, because it relies on only one cell in a chain (the end for WAM) being subject to trailing. Also note that in the Aquarius system [RD92], when trailing analysis cannot avoid the trailing of a variable, no runtime method for avoiding trailing is used instead.

The proposed analysis achieves very good speed-ups for deterministic benchmarks like hanoi_difflist and qsort_difflist. However, it can be significantly improved through, for example, the use of other kinds of information such as liveness of variables. Also, we would like to investigate the trade-off between analysis complexity and gain. Maybe a slightly less complex analysis would yield a similar speed-up. Alternatively, a more complex analysis that keeps track of the configuration of chains and terms could significantly improve speed.

References

[AK91] H. Aït-Kaci. *Warren's Abstract Machine: A Tutorial Reconstruction.* MIT Press, 1991.

[DdlBH+99a] 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.

[DdlBH+99b] 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.

[Deb92] S. Debray. A Simple Code Improvement Scheme for Prolog. *Journal of Logic Programming*, 13(1):349–366, May 1992.

[dlBMSS98] M. García de la Banda, K. Marriott, P. Stuckey, and H. Søndergaard. Differential methods in logic program analysis. *JLP*, 35:1–37, 1998.

[HCC95] P. Van Hentenryck, A. Cortesi, and B. Le Charlier. Type analysis of Prolog using type graphs. *JLP*, 22:179–209, 1995.

[JB93] G. Janssens and M. Bruynooghe. Deriving descriptions of possible value of program variables by means of abstract interpretation. *JLP*, 13:205–258, 1993.

[MWB94] A. Mulkers, W. Winsborough, and M. Bruynooghe. Live-structure dataflow analysis for prolog. *ACM TOPLAS*, 16:205–258, 1994.

[Net01] N. Nethercote. The Analysis Framework of HAL. Master's thesis, University of Melbourne, September 2001.

[Noy94] J. Noyé. *Elagage de contexte, retour arrière superficiel, modifications réversibles et autres: une étude approfondie de la WAM.* PhD thesis, Université de Rennes I, November 1994.

[RD92] P. Van Roy and A. Despain. High-performance logic programming with the aquarius prolog compiler. *IEEE Computer*, 25(1):54–68, 1992.

[SD02] T. Schrijvers and B. Demoen. Combining an Improvement to PARMA Trailing with Analysis in HAL. Technical Report 338, K.U.Leuven, April 2002.

[SHC95] 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.

[Søn86] H. Søndergaard. An application of abstract interpretation of logic programs: occur check reduction. In *European Symposium on Programming, LNCS 123*, pages 327–338. Springer-Verlag, 1986.

[Tay91] A. Taylor. *High Performace Prolog Implementation.* PhD thesis, Basser Department of Computer Science, June 1991.

[Tay96] A. Taylor. Parma - Bridging the Performance GAP Between Imperative and Logic Programming. *Journal of Logic Programming*, 29(1-3):5–16, 1996.