

Herbrand Constraint Solving in HAL

Bart Demoen*, Maria García de la Banda†, Warwick Harvey†, Kim Marriott†, and Peter Stuckey‡

* Dept. of Computer Science, K.U.Leuven, Belgium

† School of Computer Science & Software Engineering, Monash University, Australia

‡ School of Computer Science & Software Engineering, University of Melbourne, Australia

Abstract. HAL is a new constraint logic programming language specifically designed to support the construction of and experimentation with constraint solvers. One of the most important constraint solvers in any logic programming language is the Herbrand (or term) constraint solver. HAL programs are compiled to Mercury but, while Mercury provides only tests for equality and construction and deconstruction of ground terms, HAL supports full unification and logical variables. In this paper we describe the HAL Herbrand constraint solver and show how by using Parma bindings, rather than the standard term representation used in the WAM, we can implement a solver which interacts gracefully with Mercury’s term representation. Like Mercury, HAL allows type, mode and determinism declarations. HAL uses information from these declarations to reduce the overhead of Herbrand constraint solving wherever possible. We systematically evaluate the effect of each kind of declaration on the efficiency of HAL programs. Essentially, we start with a pure Prolog program, then add type, then mode, and finally determinism declarations, and compare the efficiency of these different versions. This is possible since, as HAL provides full unification and a “constrained” mode, all versions are legitimate HAL programs.

1 Introduction

Mercury is a recent logic programming language which is considerably faster than traditional Prolog implementations [7]. One reason is that Mercury requires the programmer to provide type, mode and determinism declarations. Type information allows a compact representation for terms, mode information guides reordering of literals, and determinism information removes the overhead of choice point creation. Another reason is that Mercury does not support full unification, only construction, deconstruction and equality testing for ground terms. This means that Mercury does not allow true logical variables, and so common logic programming idioms such as difference lists cannot be coded in Mercury.

Here we investigate whether it is possible to have Mercury-like efficiency, yet still allow true logical variables. We describe our experience with HAL, a new constraint logic programming language specifically designed to support the construction of and experimentation with constraint solvers [2]. Features of HAL include Mercury like declarations, a well-defined solver interface, dynamic scheduling and global variables. HAL is compiled to the logic programming language Mercury [7] so as to leverage from its low-level compilation techniques. Unlike Mercury, HAL provides a built-in Herbrand constraint solver which provides full unification and so supports logical variables.

Due to the importance of term constraints, HAL provides a built-in Herbrand constraint solver. This solver uses Parma bindings [8], rather than the standard variable representation used in the WAM [9, 1], since this allows the solver to use

essentially the same term representation for ground terms as does Mercury (see Section 3.4). This is important because it allows the HAL compiler to replace calls to the Herbrand constraint solver by calls to Mercury’s more efficient term manipulation routines whenever ground terms are being manipulated.¹

The results of our empirical evaluation of HAL and its Herbrand solver are very promising. With appropriate declarations, HAL is almost as fast as Mercury, yet allows true logical variables. And without declarations, its efficiency is comparable to that of SICStus Prolog. We also systematically evaluate the effect of each kind of declaration on the efficiency of HAL programs so as to determine where most of this speedup is coming from. This is possible since, as HAL provides full unification and a “constrained” mode, all versions are legitimate HAL programs. Our results suggest that mode declarations have the most impact on execution speed, type declarations the least and determinism declarations provide moderate speedup.

2 The HAL Language

In this section we provide a brief overview of the HAL language, concentrating on its support for Herbrand constraints. For more details see [2]. The basic HAL syntax follows the standard CLP syntax, with variables, rules and predicates defined as usual. The module system in HAL is similar to that of Mercury. The base language supports integer, float, string, atom and term data types. However, this support is limited to assignment, testing for equality, and construction and deconstruction of ground terms. More sophisticated manipulation is available by importing a constraint solver for the appropriate type.

As a simple example, the following program is a HAL version of the Towers of Hanoi benchmark which uses difference lists to build the list of moves.

```

:- module hanoi.                                (L1)
:- import int.                                  (L2)

:- typedef tower    -> (a ; b ; c).             (L3)
:- typedef move     -> mv(tower,tower).         (L4)
:- typedef list(T)  -> ([ ] ; [T | list(T)]).    (L5)
:- typedef difflist -> (list(move)-list(move)). (L6)
:- herbrand list/1.                              (L7)

:- export pred hanoi(int, list(move)).          (L8)
:-          mode hanoi(in,no) is semidet.       (L9)
hanoi(N,M) :- hanoi2(N,a,b,c,M-[ ]).            (L10)

:- pred hanoi2(int,tower,tower,tower,difflist). (L11)
:- mode hanoi2(in,in,in,in,oo) is semidet.     (L12)
hanoi2(N,A,B,C,M-Tail) :-
    ( N = 1 ->
      M = [mv(A,C)|Tail]
    ; N > 1,
      N1 is N - 1,
      hanoi2(N1,A,C,B,M-Tail1),
      Tail1 = [mv(A,C)|Tail2],
      hanoi2(N1,B,A,C,Tail2-Tail)
    ).

```

¹ Actually, as long as the term is “sufficiently” instantiated

The first line (*L1*) states that the file defines the module `hanoi`. Line (*L2*) imports the standard library module `int` which provides (ground) arithmetic and comparison predicates for the type `int`. Lines (*L3*), (*L4*), (*L5*) and (*L6*) define term types used within the module. The type `tower` gives the names of the towers, `move` defines a move as a pair of towers in a `mv` constructor, while `list` defines polymorphic lists and `difflist` defines a pair of lists of moves.

Line (*L7*) states that the Herbrand constraint solver for lists will be used in the module. We will elaborate on this later. Line (*L8*) declares that this module exports the predicate `hanoi` which has two arguments, an `int` and a list of moves. This is the *type* declaration for `hanoi`.

Line (*L9*) is an example of a *mode of usage* declaration. The predicate `hanoi`'s first argument has mode `in` meaning that it will be fixed (ground) when called, the second argument has mode `no` meaning that it will be `new` on calling (that is, never seen before) and “constrained” (`old`) on return. The second part of the declaration “`is semidet`” is a determinism statement. It indicates that `hanoi` either fails or succeeds with exactly one answer. In general, predicates may have more than one mode of usage declaration.

The rest of the file contains the rules defining `hanoi` and declarations and rules for the auxiliary predicate `hanoi2`. Note the additional mode `oo` which means the argument is “constrained” on both call and return.

2.1 Type, Mode and Determinism Declarations

As we can see from the above example, HAL allows programmers to annotate predicate definitions with type, mode and determinism declarations (modeled on those of Mercury).

Type declarations detail the representation format of a variable or argument. Types are specified using type definition statements such as those shown in (*L3*)–(*L5*). They are (polymorphic) regular tree type statements. Overloading of predicates and functions is allowed, although the definitions for different type signatures must appear in different modules. For example, in the module `hanoi` the binary function “`-`” is overloaded and may mean integer subtraction or difference list pairing.

Mode declarations associate a mode with each argument of a predicate. A mode has the form $Inst_1 \rightarrow Inst_2$ where $Inst_1$ describes the input instantiation state of the argument and $Inst_2$ describes the output instantiation state. The basic instantiation states for a solver variable are `new`, `old` and `ground`. Variable X is `new` if it has not been seen by the constraint solver, `old` if it has, and `ground` if X is constrained to take a fixed value.

For terms with more structure, such as a lists of moves, more complex instantiation states (lying between `old` and `ground`) may be used to describe the state of term. Consider, for example, the instantiation state definition:

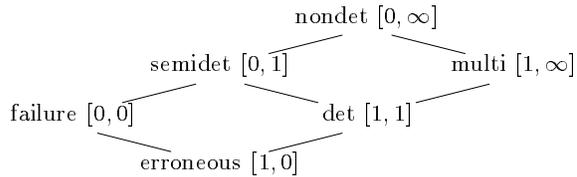
```
:- instdef old_list_of_move -> ifbound([], [ground|old_list_of_move]).
```

which is read as the variable may be unbound, but, if bound, it is either bound to an empty list or to a list with a fixed head and a tail with the same instantiation state. Another example is

```
:- instdef bound_difflist -> bound(old_list_of_move - old_list_of_move).
```

which indicates that the difference list pair is certainly constructed, but the elements in the pair may be variables. Actually, the `bound` may be dropped from the definition since this is HAL's default.

Determinism declarations detail how many answers a predicate may have. We use the Mercury hierarchy:



where the range with each declaration name gives the minimum and maximum number of solutions that it describes. The determinism **erroneous** indicates a runtime error, while **failure** indicates the predicate always fails.

The HAL compiler currently provides type checking and inference, and mode checking. Type checking and inference is performed using a propagation based constraint approach [3]. It extends Mercury’s analysis by allowing a limited form of subtyping required for natural constraint expression (though irrelevant to the Herbrand solvers). Mode checking is a relatively complex operation involving reordering body literals in order to satisfy mode constraints, and inserting initialization predicates for solver variables. It subsumes the Mercury mode checking, and handles complex partially bound instantiations like **bound_difflist**. Determinism declarations in HAL are simply passed to Mercury.

2.2 Herbrand Constraint Solvers

HAL has been expressly designed to allow experimentation with constraint solvers. The conceptual view of a constraint solver in HAL is a module that defines a type for the constrained variable, a predicate **init/1** to initialize a variable and a predicate **=/2** to equate two variables. The compiler automatically adds calls to **init/1** in user code where required, and generates calls to **=/2** because of normalization. The constraint solver will usually provide more functionality, such as other primitive constraints and solver dependent delay conditions. This view of solvers as modules facilitates “plug and play” with different solvers.

Term manipulation is at the core of any logic programming language. As indicated previously, the HAL base language only provides limited operations for dealing with terms, corresponding to those Mercury allows. Conceptually, the HAL run-time system also provides a Herbrand constraint solver for each term type defined by the programmer. This solver supports full equality (unification) and, thus, the use of true logical variables and logic programming idioms like difference lists. If the programmer wishes to make use of this more complex constraint solving for terms of some type, then they must explicitly declare that they want to use the Herbrand constraint solver for that type.

For example, in the **hanoi** module, since the programmer is using difference lists, the programmer needs to use the Herbrand constraint solver for lists. Thus, the program contains the declaration

```
:- herbrand list/1.
```

However, elements of the other types, **move**, **tower** and **difflist** are only manipulated when bound, so no **herbrand** declaration is required for those types.

Note that the **herbrand** declaration is local to a module. Thus in one module terms of the **list** type will be handled by the Herbrand solver, while in another module they may be handled by HAL’s base level term manipulation operations. Hence, the underlying term representation should support both forms of manipulation.

The main effect of the **herbrand** declaration occurs during mode checking. If the declaration is not present for a term type, then terms of that type can only

use restricted forms of the equality predicate corresponding to constructing a new term, deconstructing a bound term or comparing two bound terms. Furthermore, a new variable of that type cannot be initialized. During mode checking, literals in the body of predicates will be reordered so as to achieve the appropriate mode. If it is not possible, then a mode error results. If the **herbrand** declaration is present then terms of that type can be involved in arbitrary calls to the equality predicate.

Another way of understanding the effect of the **herbrand** declaration is in terms of the allowed instantiations for a particular type. If a term type does not have a **herbrand** declaration then non-ground terms with that type cannot be involved in an equality, unless the equality is a simple copy.

To clarify this discussion, consider the goal

```
r(X), X=[U|Z], U=1, Z=[]
```

where **X** and **Z** are lists of integers, **U** is an integer, the predicate **r** has mode **r(oo)** and at the start of the goal all variables are **new**. We examine how mode checking proceeds with and without the **herbrand** declaration for lists.

Without the declaration, **r(X)** cannot be scheduled since **X** has instantiation **new** not **old**. The constraint **X=[U|Z]** cannot be scheduled since both **Z** and **U** have instantiation **new**. **U=1** can be scheduled since it is a simple assignment. Similarly, the constraint **Z=[]** can also be scheduled. Since both **U** and **Z** are now fixed, **X=[U|Z]** can be scheduled. Since this fixes the value of **X**, the call to **r(X)** can now be scheduled. Thus the reordered goal is

```
U=1, Z=[], X=[U|Z], r(X).
```

With the **herbrand** declaration, the initial call to **r(X)** can be scheduled since the compiler can add a call to **init(X)**. The constraint **X=[U|Z]** can be scheduled since **X** is **old**. **U=1** and **Z=[]** can be scheduled since both variable are **old**. Thus the final code is

```
init(X), r(X), X=[U|Z], U=1, Z=[].
```

As we have seen instantiations in HAL can be quite powerful. However, defining such instantiations is rather laborious, especially since they are often type specific.

Typically, the HAL programmer will use the instantiation **old** for initialized variables with an associated constraint solver and **ground** if there is no associated constraint solver. For convenience, HAL allows the instantiation **old** to be used as a shorthand for the most general instantiation state of an initialized variable whose term type is “mixed”, that is, some components of the term have an associated solver (e.g. are **herbrand**), while others do not. Thus, if we return to the **hanoi** program, the **old** instantiation for variables with type **difflist** actually is a shorthand for the instantiation **bound_difflist** since **list** is Herbrand but **difflist** and **move** are not, hence, if they appear, they must be bound.

One implementation difficulty is that the same type can have a **herbrand** declaration in one module but not in another. Thus the **old** instantiation, even when applied to variables with the same type, may mean different things in different modules. To ensure that instantiations are coherent across modules, the HAL compiler expands all modes involving the **old** instantiation into their longhand equivalent.

3 Herbrand constraint solvers

Because of the importance of term constraint solving and the support of the underlying Mercury system, the HAL compiler treats Herbrand constraints specially. In this section we describe how Herbrand constraint solvers are implemented in HAL.

To begin we briefly introduce the WAM and Mercury approaches to term representation and manipulation, and then describe the Parma binding scheme of Taylor. Finally, we show how the Parma binding scheme is used to implement Herbrand constraint solvers in HAL.

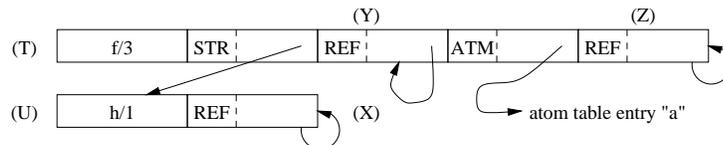
3.1 WAM

The Warren Abstract Machine (WAM) [9, 1] forms the basis of most modern Prolog implementations. We briefly review the term representation² of the WAM.

Terms are represented on a heap, which is an array of data cells. A reference cell is usually broken into two parts: a tag and a reference pointer. The most important tag values are REF (a variable reference), STR (a structure) and ATM (an atomic object e.g. a constant).

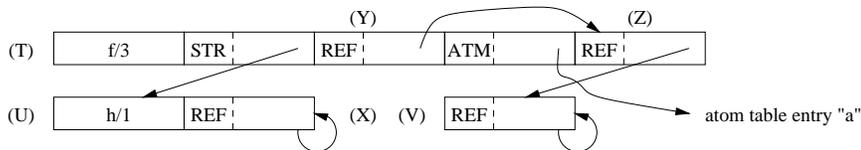
The structure $\mathbf{f}(t_1, \dots, t_n)$ is represented by a STR tagged pointer to a contiguous sequence of $n + 1$ cells. The first cell contains the functor \mathbf{f} and the arity n , and the next n cells hold the representations of t_1, \dots, t_n . An unbound variable (on the heap) points to itself. An atom is represented by a cell with tag ATM and a pointer into the atom table.

For example, the term $f(h(X), Y, a, Z)$ is stored using the representation shown below:

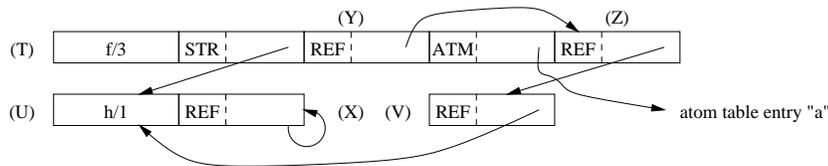


The native representation of base types such as integers and floats (usually) uses all 32 bits of the cell. WAM implementations either treat them as atoms, wrap them in a special functor, or assign tag values for the types and use the remaining bits to store the data. The disadvantage of the third approach is that less bits are available for the data itself and the tags must be removed before applying arithmetic operations.

Consider unification of two objects on the heap. First both objects are dereferenced. That is, their reference chain is followed until either a non-REF tag or a self reference is found. If at least one of the dereferenced objects is a (self) reference (i.e. an unbound variable) it is modified to point to the other object. Otherwise, the tags of the dereferenced objects are checked for equality. In the case of an ATM tag, they are checked to see they have the same atom table entry. In the case of a STR tag, the functor and arity are checked for equality, and the corresponding arguments are unified. For example, starting from the above heap state and unifying Y with the heap variable Z and then with another heap variable V results in the heap:



If we then unify Y with U (which is bound to $h(X)$) we obtain the following heap:



² For simplicity, we ignore stack variables.

Notice how references chains can exist throughout the heap.

The address of any pointer variable modified by unification is (conditionally) placed in the trail. Since the modified variable is always a self reference, its previous state can be restored from this information alone.

3.2 Mercury

The term representation of Mercury makes use of the strong static typing of the language to allow a compact type-specific representation for terms. For more details see [7]. Consider a type for lists of integers:³

```
:- typedef listint -> ([ ] ; [int | listint] ).
```

Given a term of type `listint` there are only two possibilities for its (top-level) value, it is either `nil` “[]” or `cons` “.”. Mercury reserves one tag value (`NIL`) for `nil`, and one (`CONS`) for `cons`. Since the `nil` reference does not need any further information the pointer part is 0. A `cons` structure is simply two contiguous cells: the first is an integer and the second is a reference to the rest of the list.

Assuming 32 bit words and aligned addressing the low two bits of a pointer are zero. In Mercury these bits are used for storing the tag values, hence 4 different tags are available. For term types with more than 4 functors, the representation is modified. Since for a constant tag (such as `NIL`) the remaining part of the cell is unused, the remaining 30 bits can be used to store different constant functors. For term types with more than 4 non-constant functors, the Mercury representation uses an extra cell to store the identity of the functor, much like the WAM representation (although the arity of the functor does not need to be stored since the type information gives this).

Mercury does not allow true logical variables, so there is no analogue to the `REF` tagged references used in the WAM. Because there are no references and types are known at compile time, there is no need to have tags for every object. Hence, an object of a primitive type, like an integer, is free to use its entire 32 bit cell to store its value.

Mercury performs program normalization, so that only two forms of equations are directly supported: $X = Y$ and $X = f(A_1, \dots, A_n)$ for each functor f where A_1, \dots, A_n are distinct variables.

The equation $X = Y$ is only valid in two modes. The first is when X is fixed (i.e. has instantiation `ground`) and Y is new (i.e. has instantiation `new`), or vice versa. In this case the fixed variable is copied into the new. The second is when both X and Y are fixed in which case a procedure to check the two terms are identical is called. Mercury generates a specialized procedure (we’ll refer to as `unify_gg`) to do this for each type.

The equation $X = f(A_1, \dots, A_n)$ is only valid in three modes. The first is when X is new and A_1, \dots, A_n are all fixed. In this case a contiguous block of n cells is allocated, the values of A_1, \dots, A_n copied into these cells, and X is set to a pointer to this block with an appropriate tag. The second is when X is fixed and each A_1, \dots, A_n is new, in which case after testing that X is of the appropriate form, the values in the contiguous block of n cells that it points to are copied into A_1, \dots, A_n . The third is when all involved variables are fixed, in which case the specialized procedure is used to check that they are identical. The case where some of A_1, \dots, A_n are new and some fixed (e.g. A_4) is handled by replacing each such variable in the equation by a new variable (e.g. A'_4) and a following equation (e.g. $A'_4 = A_4$).

As an example, consider how Mercury will (attempt) to compile the equation, $T = f(h(1), Y, a, Y)$ where Y and T are new. First it is normalized to give the

³ We use HAL syntax for uniformity rather than that of Mercury.

equations

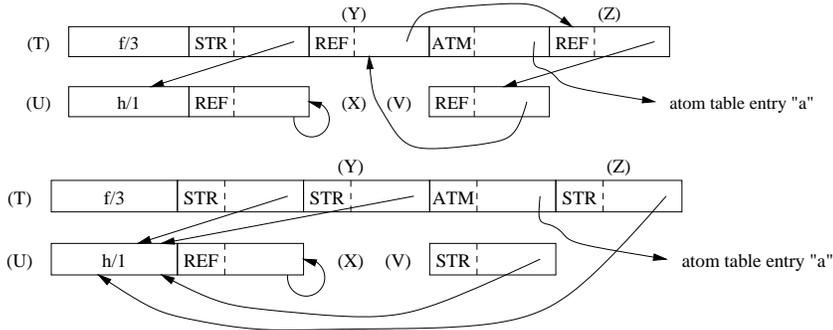
$$X = 1, U = h(X), S = a, Z = Y, T = f(X, Y, S, Z)$$

The first three equations can be compiled to “construct” the variable on the lefthand side. The two remaining equations cannot be compiled since they do not satisfy one of the above modes. If later in the goal Y is given a fixed value then these two equations can be compiled to construct the lefthand side variable.

3.3 Parma variable representation

In the Parma system [8], Taylor introduced a new technique for handling variables, that avoided the need for dereferencing potentially long chains when checking whether an object is bound or not. A non-aliased free variable on the heap is still represented as a self-reference like in WAM. The difference occurs when two free variables are unified. Rather than pointing one at the other, as in the WAM, a cycle of bindings is created. In general n variables which are all equal are represented by n cells forming a cycle. When one of the variables is equated to a non-variable all of the variables in the cycle are changed to direct (tagged) pointers to this structure and changes are trailed.

For example, the corresponding Parma heap structures for the last two diagrams of WAM heaps are shown below.



Notice that the chain of references is now a cycle. After processing the constraint $Y = U$ each variable in the cycle is replaced by a direct reference to the heap representation for $h(X)$.

The Parma scheme for variable representation has the advantage that dereferencing of terms on the heap is never required. However, it has a number of potential disadvantages:

- Checking if two unbound variables are equivalent is more involved, and is required for variable-variable binding. Essentially, each variable’s cycle of aliased variables may need to be traversed.
- When instantiating a variable cycle (conditional) trailing must occur for each cell in the cycle (rather than one as for WAM), and the trail requires two words (the variable position and its old value).
- When creating a structure that will hold a copy of an already existing unbound variable, the cycle of variables grows, and trailing potentially occurs.

The impact of each of these factors is dependent on the length of the cycles that are manipulated. As we shall see, cycles rarely grow beyond 1.

Note that only heap variables can be placed in a variable’s alias cycle. An unbound initialized variable on the stack or in a register points into a cycle on the heap. If this cycle is then bound, the stack or register variable becomes a pointer to a bound object. This means that when accessing data through a stack variable or register, the Parma scheme sometimes requires a single step dereference.

3.4 HAL

Recall that HAL allows the same term to be used with a Herbrand solver and with HAL's basic term manipulation facilities. Since HAL is compiled into Mercury, it makes considerable sense for HAL's basic term manipulation facilities to be directly implemented by those of Mercury. Therefore HAL's Herbrand solvers need to use a term representation which is compatible with that of Mercury.

HAL employs the Parma approach to variable binding with the Mercury term representation scheme. The main reason for using the Parma approach, rather than that of the WAM, is that when a term structure becomes ground in the Parma scheme it has no reference chains within it. Hence, it becomes a legitimate Mercury term. Even when a term is only partially bound, the HAL compiler can (mis)use the efficient Mercury operations to manipulate the bound part of the term, since they will still give the desired behavior.

HAL reserves the tag 0 for all term types for use as the REF tag. This means that instead of 4 tags generally available for representing a type there are only 3.

During compilation HAL (like Mercury) normalizes programs so that only two forms of equations arise: $X = Y$ and $X = f(A_1, \dots, A_n)$ (where each A_i is distinct). The compiler translates these equations into calls to appropriate Mercury and C code to implement the Parma variable scheme.

Consider an equation $X = Y$. If one of X and Y is new and one is old, we need simply point the new variable at the old, and Mercury handles it correctly. When both X and Y are new an initialization `init(Y)` is added beforehand. The initialization allocates a new cell, makes it a self-pointer and returns a reference to this cell in Y . This makes Y old and the previous case applies. If both X and Y are fixed we call Mercury's specialized procedure `unify_gg`.

The only remaining case, where both X and Y are not new but are not both completely ground, corresponds to a true unification. We replace this with a call to the Herbrand unification procedure `unify_oo` automatically generated by HAL for the type of X and Y . It has the form:

```
:- pred unify_oo(t,t).
:- mode unify_oo(oo,oo).
unify_oo(X,Y) :-
    (var(X) ->    (var(Y) -> unify_var_var(X,Y)
                  ;        unify_var_val(X,Y))
    ;            (var(Y) -> unify_var_val(Y,X)
                  ;        unify_val_val(X,Y))).
```

The `unify_val_val` is similar to the Mercury procedure `unify_gg` except it calls `unify_oo` on the arguments of unified terms rather than `unify_gg`. The `var` procedure simply checks the tag of its argument is zero.

The procedure `unify_var_var` shown in Figure 1 unifies two variables. It first checks that the variables are not already the same, and then joins the cycles together, trailing the change. Note that, unlike the case for WAM, the code for unifying two variables is symmetric, treating each variable the same way. The procedure `unify_var_val` shown in Figure 1 unifies a variable and a non-variable. It modifies all the variables in the cycle to directly refer to the non-variable, and trails the changes.

Processing an equation of the form $X = f(A_1, \dots, A_n)$ is more complicated since we may have to create objects on the heap. The first case, when X is new, will create a new structure on the heap. Each variable A_i must not be new, unless their type is `herbrand`. New variables of `herbrand` types are initialized during the building of the heap structure. Assume that variables A_{o_1}, \dots, A_{o_m} are `herbrand`

```

unify_var_var(X,Y) =
  BeginX = X; BeginY = Y;
  QX = *X; QY = *Y;
  while (QX != BeginY && QY != BeginX)
    if (QX != BeginX && QY != BeginY)
      { QX = *QX; QY = *QY; }
    else
      { trail(BeginX); trail(BeginY);
        Tmp = *BeginX;
        *BeginX = *BeginY;
        *BeginY = Tmp;
        break; }

unify_var_val(X,Y) =
  BeginX = X; QX = X;
  repeat
    { Next = *QX;
      trail(QX);
      *QX = Y;
      QX = Next; }
  until (QX == BeginX)

```

Fig. 1. Psuedo-C code for HAL unification

and old, and A_{n_1}, \dots, A_{n_l} are **herbrand** and new. The translation to Mercury is simply:

```

dummy_init(An1), ..., dummy_init(Anl),
X = f(A1, ..., An),
An1 = init_heap(X, n1), ..., Anl = init_heap(X, nl),
fix_copy(X, o1), ..., fix_copy(X, om)

```

The `dummy_init(A)` procedure is used to make Mercury believe it can schedule the construction but contains no code; subsequent in-lining by Mercury removes it. The `init_heap(X, i)` procedure creates a self reference in the i^{th} slot of heap cells pointed to by X and returns it:

```

init_heap(X, i) { return X[i] = &(X[i]); }

```

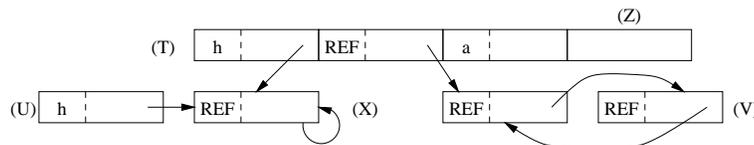
The `fix_copy(X, ok)` procedure handles the case when A_{o_k} is a reference, which was copied by Mercury into the new heap structure. If A_{o_k} was an unbound variable this results in a reference to the cycle in the o_k^{th} cell rather than o_k^{th} cell being placed in the cycle. The procedure adds the o_k^{th} cell into the cycle. If A_{o_k} is bound but not dereferenced (this can happen for stack and register variables) the procedure replaces the contents of the o_k^{th} cell by what it refers to.

```

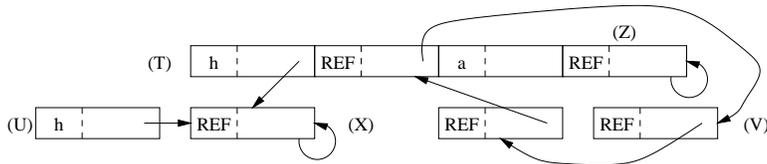
fix_copy(X, i)
  AXi = &(X[i]); Xi = X[i];
  if (var(Xi))
    if (var(*Xi))
      { trail(Xi);
        *AXi = *Xi; *Xi = AXi }
    else *AXi = *Xi;

```

For example, consider the construction of $T = f(U, V, S, Z)$ where T and Z are new and U is known to be bound (to $h(X)$), S is known to be bound (to a) and V is old (and part of a cycle). After executing the Mercury $X = f(U, V, S, Z)$ the heap looks like:



Then applying `init_heap(T,4)` and `fix_copy(T,2)` results in the heap:



The second case for handling an equation of the form $X = f(A_1, \dots, A_n)$ occurs when X is known to be bound and A_1, \dots, A_n are new. This is simply left to Mercury. If one (or more) of A_1, \dots, A_n are not new, they are replaced by new variables and equations as in the Mercury case. The third case is all variables involved are fixed in which case it uses the Mercury code. when X is old. The final case is when X is old. The generated code checks if X is bound in which case it treats the equation as if it were the deconstruction $X = f(B_1, \dots, B_n)$ followed by equations $A_i = B_i$. Otherwise X is a variable and the code constructs the term $f(A_1, \dots, A_n)$ on the heap⁴ and the equates X to this term using `unify_oo`.

As in the Parma system, only heap variables can be placed in a variable's alias cycle. Thus a stack variable or register must be a pointer somewhere into the cycle and so when accessing data through a stack variable or register HAL sometimes requires a single step dereference. Consider compiling the following code:

```
:- herbrand listint.
:- pred p(listint).
:- mode p(bound([ground|old])->bound([ground|old])).
p(Z) :- Z = [N|_].
goal :- X = Z, X = [4|_], p(Z).
```

An automatic initialization of Z is added to `goal` during mode checking. During the execution of `goal` (assuming reordering is switched off) the registers holding X and Z and the heap change as illustrated in Figure 2. Note that for the call to `p(Z)`, Z must have a one step dereference. Inside `p(Z)`, the deconstruct is simply a Mercury deconstruct.

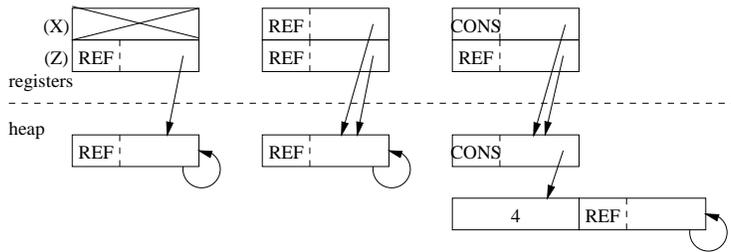


Fig. 2. Register and heap representation for each stage of `init(Z)`, `X = Z`, `X = [4|_]`.

One important issue we have not discussed is the treatment of variables which have a polymorphic type. HAL produces a general unification predicate `unify_oo` for each type, and polymorphic code is passed the appropriate version in the run-time type information. At present, the compiler does not support the use of Herbrand types for polymorphic arguments, e.g. `list(T)` where type T is `herbrand`. Handling this case requires adding additional run-time type information to Mercury and has not been implemented so far.

⁴ Depending on whether arguments are `herbrand` or not this may not be possible, causing a run-time error.

4 Evaluation

This section has three aims. The first is to compare the performance of HAL and its Herbrand solver with a state-of-the-art Prolog implementation, SICStus Prolog. The second is to investigate the impact of each kind of declaration on efficiency. The third is to compare HAL with Mercury so as to determine the overhead introduced by the run-time support for logical variables and the Herbrand solver.

To achieve the first aim we take a number of Prolog benchmarks and transform built-ins not present in HAL (such as `cut`) into their HAL equivalents (such as `if-then-else`). We then compare these with the equivalent HAL program. Although Prolog does not have type, mode and determinism declarations, the current HAL compiler requires them. The HAL program is obtained by defining a “universal” term type for the program which contains all functors occurring in the program and declaring this type to be `herbrand`. All predicate arguments are declared to have this type and mode `oo`, and all predicates are declared to have determinism `nondet`. All integers, floats, atoms and strings in the program must be wrapped, and each wrapping functor must appear in the “universal” term type.

Most of these tasks are done automatically by a pre-processor. For example, for the original `hanoi` Prolog program (the code in Section 2 minus the declarations), the preprocessor will add the declarations

```
:- typedef htype -> (int(int) ; float(float) ; atom(atom) ; char(char)
                    ; [htype|htype] ; mv(htype,htype) ; htype - htype ).
:- herbrand htype.
:- pred hanoi(htype,htype).
:- mode hanoi(oo,oo) is nondet.

:- pred hanoi2(htype,htype,htype,htype,htype).
:- mode hanoi2(oo,oo,oo,oo,oo) is nondet.
```

It will also replace the three occurrences of `1` in the program text by `int(1)`, replace `[]` by `atom([])` and create predicates for the wrapped versions of `>`, `is` and function `-`.

To achieve the second aim we take these Prolog-equivalent HAL programs and add precise type, then mode, then determinism declarations, and compare the resulting efficiencies. Note that we must continue to wrap integers and other primitive types even when we add type declarations since Prolog sometimes treats them as logical variables. For example, integer variables may be placed in data structures or equated before they are fixed. It is only when mode information is available that we can unwrap them. All types introduced in this step must be declared to be `herbrand`. Once mode declarations are added, types which never have the `old` instantiation need not be declared as `herbrand` and, in the case of the primitive types, can have their wrapping removed.

To achieve the last aim, we run the HAL programs with precise declarations on a corresponding version of Mercury that does not provide trailing or use the extra `REF` tag. Since Mercury does not provide full unification, we can only do this for benchmarks without a `herbrand` declaration.

Table 4 details our comparison. We have turned garbage collection off in all three systems: SICStus Prolog 3.7.1, Mercury May 1999 HAL development version, and HAL. All timings are in seconds on a dual Pentium II-400MHz with 384M of RAM running under Linux-2.2.

We have used a subset of the standard Prolog benchmarks: `aiakl`, `boyer`, `deriv`, `fib`, `mmatrix`, `serialize`, `tak`, `warplan`, `hanoi` and `qsort`. The last two are shown in two forms, one using “normal” lists and `append`, the other using difference lists. The reason for choosing these benchmarks is that they did not require extensive

Benchmark	Size		SICStus		HAL				Merc
	Preds	Lits	Orig	Mod	No Info	T	T+M	T+M+D	Merc
aiakl	7	21	0.09	0.08	0.29	0.25	0.04	0.03	0.03
boyer	14	124	0.69	0.69	2.11	1.80	0.09	0.05	0.04
deriv	1	33	2.11	3.03	2.90	2.94	0.74	0.50	0.31
fib	1	6	1.46	1.45	0.25	0.20	0.02	0.02	0.01
hanoiapp	2	7	3.42	3.43	3.57	3.23	0.61	0.22	0.23
hanoiff	2	6	2.34	2.34	0.27	0.24	0.08	0.11	na
mmatrix	3	7	1.29	1.29	1.28	0.96	0.11	0.04	0.04
qsortapp	3	10	4.04	2.33	2.98	2.56	0.39	0.19	0.14
qsortdiff	3	10	3.94	2.26	2.87	2.58	0.37	0.21	na
serialize	5	19	1.78	1.77	1.28	1.04	0.35	0.27	na
tak	1	9	0.77	0.95	0.74	0.53	0.10	0.05	0.03
warplan	25	88	0.59	0.82	1.55	0.45	0.33	0.22	na

Table 1. Execution times in seconds

changes to the original Prolog benchmarks⁵ and hence the comparison is fairer. The second and third columns of Table 4 detail the benchmark sizes (number of predicates and literals before normalization, excluding dead code and the query). Subsequent columns give the execution time for

- the original program run with SICStus Prolog,
- the modified Prolog program run with SICStus Prolog,
- the Prolog-equivalent HAL program (the one obtained with the preprocessor),
- with precise type definitions,
- with precise type and mode declarations,
- with precise type, mode and determinism declarations,
- this last version run with Mercury (if possible).

In general, the original and modified SICStus programs have the same speed. `deriv` slows down because of loss of indexing, while the two versions of quick sort improve because a badly placed cut in the original program is replaced by a more efficient if-then-else. The Prolog-equivalent HAL versions have roughly similar efficiency to the modified SICStus versions. Slow-down occurs in `aiakl`, `boyer` and `warplan` because no indexing is currently available for non-bound input arguments. Surprising speed-up occurs for `fib` and `hanoiff`; we suspect because of Mercury’s handling of recursion. Generally, adding precise type information leads to a significant improvement (on average 1.4 times faster). For `warplan` the improvement is significant because it allows a type specialized version of `univ` to be used. Adding mode declarations provides the most speed-up (on average 6.8 times faster). This is because it allows calls to the Herbrand solver to be replaced by calls to Mercury’s specialized term manipulation operations and also allows indexing. Determinism declarations also lead to significant speed-up (on average 1.7 times faster). The comparison between Mercury and HAL running the version with precise declarations shows that the performance is comparable although there is overhead for supporting logical variables (we believe primarily because of support for trailing). Note that the comparison is not made for `hanoiff`, `qsortdiff`, `serialize` or `warplan` since their final HAL versions contain `herbrand` declarations.

⁵ `aiakl`, `deriv`, `qsort`, `serialize` and `tak` only required replacement of cuts by if-then-else while `warplan` also needed to transform the `\+` built-in into an if-then-else and include a well-typed version of `univ` for `warplan`. The only exception is `boyer`, for which the starting point was a restricted Mercury version, rather than the Prolog one.

We have also investigated the effect of the declarations on memory usage. Adding precise type definitions reduces the size of the stack for non-deterministic predicate calls and the heap. Adding precise mode declarations generally reduces the heap size and greatly reduces trail size—only those benchmarks with **herbrand** declarations may need to use the trail.

Finally, we have investigated the size of the alias cycles constructed using Parma bindings. Virtually all cycles have length one immediately before being bound to a non-variable term. Only two benchmarks, **warplan** and **serialize**, have a maximum cycle length of more than 2 (4 and 24 respectively).

5 Related work

As far as we know, HAL is the first logic programming implementation to use the Parma variable representation and binding scheme since this was introduced in [8]. No experimental comparison of the Parma binding scheme with that of WAM exists, although [4] discusses in detail the differences between the two. There seems to be no compelling reason to prefer one over the other; in fact, artificial examples can be constructed for which each scheme easily outperforms the other.

There has been some earlier work on the impact of type, mode and determinism information on the performance of Prolog, but the results are quite uneven. In [5] information about mode, type and determinism is used to (manually) generate better code. Its results show up to a factor of 2 speedup for mode information, and the same result for type information. [6] describes Aquarius, a Prolog system in which compile-time analysis information (including type, mode and determinism information) is used for optimizing the execution. In its results, analysis information had a relatively low impact on speed: on average about 50% for small programs without built-ins (for **tak** a factor 3) and about 12% for larger programs with built-ins (for **boyer** only 3%). Finally, in the context of the Parma system, [8] also reports on speedup obtained from information provided by compile time analysis. Its results are highly benchmark dependent, with only 10% speed up for **boyer** but a factor of 8 for **nrev**.

It is difficult to directly compare our results with those found for Aquarius and Parma. One difference is that their information is obtained from compile time analysis, rather than from programmer declarations. We suspect that compile time analysis is not powerful enough to find accurate information about the larger benchmarks, while in our experiments the programmer provides this information. This would explain why our performance improvements are more uniform (and larger) across all benchmarks, regardless of size. The second issue is the differences between the underlying abstract machines. For instance, Mercury performs particular optimizations like specializing the tags per type, the use of a separate stacks for deterministic and nondeterministic predicates and a middle-recursion optimization, which are not found in Parma or Aquarius. On the other hand, Mercury lacks real last call optimization. However, in accord with our findings, for all systems mode information gives greater speedups than type information.

6 Conclusions

We have demonstrated that it is possible to combine Mercury-like efficiency for ground data structure manipulation with Prolog-style logical variables. By using Parma bindings we ensure that the representation for terms used inside HAL's Herbrand solver is consistent with that used by Mercury for ground terms. This means that the compiler is free to use the more efficient Mercury term manipulation operations whenever this is possible.

The performance of HAL has been very pleasing. Much of this arises from the sophisticated compilation techniques used by the underlying Mercury compiler. Indeed, we have only employed the Mercury compiler using the standard optimizations. Mercury also provides other more advanced optimizations which we can also leverage from. In addition there are a number of ways to improve HAL's Herbrand constraint solving which we shall investigate: better tracking of where one-step dereferencing may be required, more specialized cases for equality and indexing for old terms.

Prolog-like programs written in HAL run somewhat slower than in SICStus, in part because there is no term indexing for non-bound instantiations. However, once declarations are provided the programs run an order of magnitude faster. Our experimental results show that the biggest performance improvement arises from mode declarations, closely followed by that for determinism. Type declarations give little speed improvement but do reduce the space requirements and are required before using either mode or determinism declarations.

It should be remembered that declarations are not only useful for improving efficiency. They also allow compile time checking to improve program robustness, help program debugging and facilitate integration with foreign language procedures.

Acknowledgements

Many people have helped in the development of HAL. In particular the Mercury development team, especially Fergus Henderson and Zoltan Somogyi, have helped us with many modifications to the Mercury system to support HAL.

References

1. H. Ait-Kaci. *Warren's Abstract Machine*. MIT Press, 1991.
2. B. Demoen, M. García de la Banda, W. Harvey, K. Marriott, and P. Stuckey. An overview of HAL. Technical Report 99/17, University of Melbourne, Dept of CS&SE, 1999.
3. B. Demoen, M. García de la Banda, and P.J. Stuckey. Type constraint solving for parametric and ad-hoc polymorphism. In J. Edwards, editor, *Proceedings of the 22nd Australian Computer Science Conference*, pages 217–228. Springer-Verlag, January 1999.
4. Thomas Lindgren, Per Mildner, and Johan Bevemyr. On Taylor's scheme for unbound variables. Technical report, UPMAIL, October 1995.
5. André Mariën, Gerda Janssens, Anne Mulkers, and Maurice Bruynooghe. The impact of abstract interpretation: an experiment in code generation. In *Proc. of the Sixth International Conference on Logic Programming*, pages 33–47. MIT Press, 1989.
6. P. Van Roy. Can Logic Programming Execute as Fast as Imperative Programming? Report 90/600, UCB/CSD, Berkeley, California 94720, Dec 1990.
7. Zoltan Somogyi, Fergus Henderson, and Thomas Conway. The execution algorithm of Mercury: an efficient purely declarative logic programming language. *Journal of Logic Programming*, 29:17–64, 1996.
8. A. Taylor. PARMA—bridging the performance gap between imperative and logic programming. *Journal of Logic Programming*, 29(1–3), 1996.
9. D. H. D. Warren. An abstract prolog instruction set. Technical Report 309, SRI International, Menlo Park, U.S.A., Oct. 1983.