Herbrand Constraint Solving in HAL

Bart Demoen
Dept. of Computer Science, K.U.Leuven, Belgium

María García de la Banda, Warwick Harvey, Kim Marriott
School of Comp. Sci. & Soft. Eng., Monash University, Australia

Peter J. Stuckey
Dept. of Comp. Sci. & Soft. Eng., University of Melbourne, Australia

Abstract

HAL is a new constraint logic programming language specifically designed to support 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 assignment, tests for equality and construction and deconstruction of ground terms, HAL supports full unification (without the occurs check). 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 is compatible with Mercury's term representation. Like Mercury, HAL supports type, mode and determinism declarations. HAL uses information from these declarations to reduce the overhead of Herbrand constraint solving wherever possible. In this paper we also systematically evaluate the effect of each kind of declaration on the efficiency of HAL programs.

1 Introduction

Mercury [6] is a recent logic programming language which is considerably faster than traditional Prolog implementations. 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 multivariant specialization while determinism information is used to remove the overhead of unnecessary choice point creation. Another reason is that Mercury does not support full unification; only assignment, 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 (directly) 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 Mercury so as to leverage from Mercury's compilation techniques. Unlike Mercury, HAL includes a built-in Herbrand constraint solver which provides full unification (without the occurs check), thus supporting logical variables. The solver uses PARMA bindings [7], rather than the standard variable representation used in the WAM [1, 8], 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 HAL provides full unification and a "constrained" mode. Our results suggest that mode declarations have the most impact on execution speed while type and determinism declarations provide moderate speedup.

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 [7]. We note that [3] discusses in detail the differences between the PARMA and WAM schemes. However, there seems to be no compelling reason to prefer one over 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 [4], information about type, mode and determinism is used to (manually) generate better code. Its results show up to a factor of two speedup for mode information, and the same result for type information. [5] 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` 300%) and about 12% for larger programs with built-ins (for `boyer` only 3%). Finally, in the context of the PARMA system, [7] 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 (from Section 4) with those found for Aquarius and PARMA. One problem is the differences between the underlying abstract machines and the optimizations performed by each compiler. Another problem is that their information is obtained from compile time

[1] Actually, as long as the term is "sufficiently" instantiated.

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.

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 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)
:- export typedef tower    -> (a ; b ; c).                        (L3)
:- export typedef move     -> mv(tower,tower).                    (L4)
:- export 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) ).
```

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 in this module. The type `tower` gives the names of the towers, `move` defines a move as a pair of towers wrapped in a `mv` constructor, `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/2` which has two arguments, an `int` and a list of moves. This is the *type* declaration for `hanoi/2`. Line ($L9$) is an example of a *mode of usage* declaration. The predicate `hanoi/2`'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/2` either succeeds with exactly one answer or fails. In general, predicates may have more than one mode of usage declaration. The rest of the file contains the rules defining `hanoi/2` and declarations and rules for the auxiliary predicate `hanoi2/5` (the mode `oo` 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 (modelled 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 ($L3$)–($L6$). 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 list of moves, more complex instantiation states (lying between `old` and `ground`) may be used to describe the state. 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: `nondet` means any number of solutions; `multi` at least one solution; `semidet` at most one solution; `det` exactly one solution; `failure` no solutions; and `erroneous` a runtime error.

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 may generate calls to `=/2` through 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, they need 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 they don't require `herbrand` declarations.

Note that the `herbrand` declaration is local to a module. Thus, in one module terms of the `list` type may 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 assignment, 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.

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/1` 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`. Obviously both `U=1` and `Z=[]` can be scheduled since they are simple assignments. These constraints fix `U` and `Z`, so then `X=[U|Z]` can be scheduled. This in turn fixes the value of `X`, so the call to `r(X)` can 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 immediately 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 variables 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 can be 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 declared `herbrand` but `difflist` and `move` are not. Hence, terms of type `difflist` will be either `new` or bound.

3 Herbrand constraint solvers

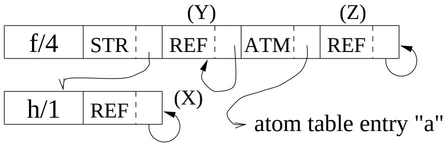
In this section we describe how Herbrand constraint solvers are implemented in HAL. We briefly introduce the WAM and Mercury approaches to term representation and manipulation, as well as describe the PARMA binding scheme of Taylor. Then we show how the PARMA binding scheme is used to implement Herbrand constraint solvers in HAL.

3.1 WAM

The Warren Abstract Machine (WAM) [8, 1] forms the basis of most modern Prolog implementations. Terms are stored on a heap,[2] 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).

[2]For simplicity, we ignore stack variables.

The structure $\mathtt{f}(t_1, \ldots, t_n)$ is represented by a STR tagged pointer to a contiguous sequence of $n+1$ cells. The first cell contains the functor $\mathtt{f}$ and the arity n, and the next n cells hold the representations of t_1, …, 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 $f(h(X), Y, a, Z)$ may be represented by:

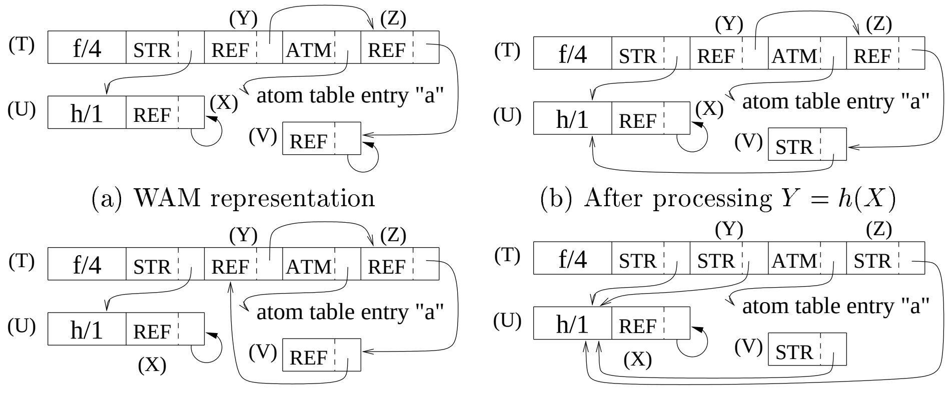


The native representation of base types such as integers and floats (usually) uses the entire width of the cell. WAM implementations either treat them as atoms, wrap them in a special functor, or assign tag values for the types and use the remaining bits to store the data.

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, if they are, 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 shown in Figure 1(a). If we then unify Y with U (which is bound to $h(X)$) we obtain the heap shown in Figure 1(b). Notice how references chains can exist throughout the heap.

The address of any pointer variable modified by unification is (condition-



(a) WAM representation (b) After processing $Y = h(X)$

(c) PARMA representation (d) After processing $Y = h(X)$

Figure 1: Different term and variable binding schemes

ally) 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 PARMA variable representation

In the PARMA system [7], 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 as in the 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 PARMA heap structures corresponding to Figures 1(a) and (b) are shown in Figures 1(c) and (d), respectively. Note that the chain of references is now a cycle. After processing $Y = U$ 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 bound terms on the heap is never required. However, it has three potential disadvantages: (a) 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. (b) 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). (c) 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 length one.

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.

3.3 Mercury

The strong static typing of Mercury allows it to use a compact type-specific representation for terms. For more details see [6]. 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 four different tags are available. For term types with more than four functors, the representation is modified. Since for a constant functor (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 non-constant functors than remaining tags, the Mercury representation uses an extra cell to store the identity of the extra functors, much like the WAM representation. In what follows, we will ignore this for simplicity.

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 cell to store its value.

Mercury performs program normalization, so that only two forms of equations are directly supported: $X = Y$ and $X = f(A_1, \ldots, A_n)$ for each functor f where $A_1, \ldots, A_n$ 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 it as `unify_gg`) to do this for each type.

The equation $X = f(A_1, \ldots, A_n)$ is only valid in three modes. The first is when X is new and A_1, ..., A_n are all fixed. In this case a contiguous block of n cells is allocated, the values of A_1, ..., 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, ..., 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, ..., 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, ..., 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. $A'_4 = A_4$).

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 should 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 four tags generally available for representing a type there are only three.

During compilation HAL (like Mercury) normalizes programs so that only two forms of equations arise: $X = Y$ and $X = f(A_1, \ldots, A_p)$ (where each A_i is a distinct variable). 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 simply need to assign the old variable to the new; Mercury handles this correctly. When both X and Y are new an initialization `init(Y)` is added beforehand. The initialization allocates a new cell on the heap, 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 old, corresponds to a true unification. We replace this with a call to the Herbrand unification procedure `unify_oo`, which is automatically generated by HAL for the type `t` of X and Y. It has the form:

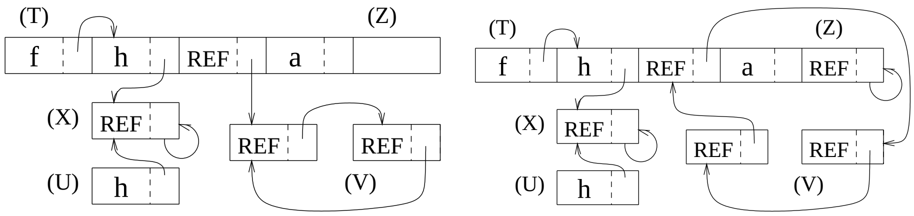
```
:- pred unify_oo(t,t).
:- mode unify_oo(oo,oo).
unify_oo(X,Y) :- (var(X) -> (var(Y) -> unify_var_var(X,Y)
                             ;           unify_var_val(X,Y))
                 ;           (var(Y) -> unify_var_val(Y,X)
                             ;           unify_val_val(X,Y))).
```

`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 if its argument has tag zero (i.e. is unbound).

The procedure `unify_var_var` shown below 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` (also shown below) 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.

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

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



(a) After Mercury construct (b) Corrected version

Figure 2: Adapting Mercury's term construction for Herbrand terms

Processing an equation of the form $X = f(A_1, \ldots, A_p)$ 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 its type is **herbrand**. New variables of **herbrand** types are initialized during the building of the heap structure. Assume that variables $A_{o_1}, \ldots, A_{o_m}$ are **herbrand** and old, and $A_{n_1}, \ldots, A_{n_l}$ are **herbrand** and new. The translation to Mercury is essentially:

dummy_init(A$_{n_1}$), ..., dummy_init(A$_{n_l}$),
X = f(A$_1$, ..., A$_p$),
A$_{n_1}$ = init_heap(X,$n_1 - 1$), ..., A$_{n_l}$ = init_heap(X,$n_l - 1$),
fix_copy(X,$o_1 - 1$), ..., fix_copy(X,$o_m - 1$)

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

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

The fix_copy($X, o_k - 1$) 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 simple copy results in a reference to the cycle in the o_k^{th} cell rather than the o_k^{th} cell being placed in the cycle. fix_copy 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) fix_copy replaces the contents of the o_k^{th} cell by what it refers to. It is defined as:

```
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 $T = f(U, V, S, Z)$ the heap is as shown in Figure 2(a). Applying init_heap(T,3) and fix_copy(T,1) gives the heap shown in Figure 2(b).

The second case for handling an equation of the form $X = f(A_1, \ldots, A_p)$ occurs when X is known to be bound and $A_1, \ldots, A_p$ are new. This is simply left to Mercury. If one (or more) of $A_1, \ldots, A_p$ are not new, they are replaced by new variables and equations as in the Mercury case. The third case is when all variables involved are fixed; again this is left to Mercury. 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 $X = f(B_1, \ldots, B_p)$ followed by equations $A_i = B_i$. Otherwise X is a variable and the code constructs the term $f(A_1, \ldots, A_p)$ on the heap[3] and then 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.

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 is conceptually straightforward, but has not yet been implemented.

4 Evaluation

Our empirical evaluation 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 Herbrand solving.

To achieve the first aim we take a number of Prolog benchmarks[4] 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. We solve this problem by defining a "universal" term type for the HAL 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

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

[4] See `www.csse.monash.edu.au/~mbanda/hal`.

```
:- typedef htype -> (int(int) ; float(float) ; atom(atom)
                    ; [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 `[]`, `a`, `b` and `c` by `atom([])`, `atom(a)`, `atom(b)` and `atom(c)`, and create predicates for the wrapped versions of `>`, `is` and function `-`.

To investigate the impact of each kind of declaration on efficiency we take these Prolog-equivalent HAL programs and add precise type, then mode, then determinism declarations, and compare the resulting efficiencies. In the first step type declarations are added. All types in this step must be declared as `herbrand` since the associated terms may sometimes be treated as logical variables. This also implies that we must continue to wrap integers and other primitive types since they may be placed in data structures or equated before they are fixed. In the second step 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. Finally, determinism declarations are added.

To compare HAL with Mercury, we run Mercury versions of the programs (with precise declarations) on a version of Mercury that does not provide trailing or reserve the extra REF tag, but is otherwise equivalent to the one used for compiling the HAL program. Since Mercury does not provide full unification, we can only do this for benchmarks with no remaining `herbrand` declarations.

Table 1 details our comparison. We have turned garbage collection off in all three systems: SICStus Prolog 3.7.1 (compact code), Mercury (May 1999 HAL development version), and HAL. All timings are in seconds on a dual Pentium II-400MHz with 384M of RAM running 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/3`, the other using difference lists. The reason for choosing these benchmarks is that they did not require extensive changes to the original Prolog benchmarks (except for `boyer`, for which the starting point was a restricted Mercury version, rather than the Prolog one) and hence the comparison is fairer. The second and third columns of Table 1 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 (obtained with the preprocessor),
- with precise type definitions,
- with precise type and mode declarations,

	Size		SICStus		HAL				Merc
Benchmark	Preds	Lits	Orig	Mod	None	T	+M	+D	
`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
`hanoidiff`	2	6	2.34	2.34	0.27	0.24	0.08	0.11	—
`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	—
`serialize`	5	19	1.78	1.77	1.28	1.04	0.35	0.27	—
`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	—

Table 1: Execution times in seconds

- 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. Slowdown occurs in `aiakl`, `boyer` and `warplan` because no indexing is currently available for possibly unbound input arguments. Surprising speed-up occurs for `fib` and `hanoidiff`; 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 very large 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 `hanoidiff`, `qsortdiff`, `serialize` or `warplan` since their final HAL versions still need `herbrand` declarations.

We have also investigated the effect of the declarations on memory usage. Adding precise type definitions reduces the stack sizes for non-deterministic predicate calls and the heap size. 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 two (4 and 24 respectively).

Our empirical evaluation of HAL is very pleasing. It demonstrates that it is possible to combine Mercury-like efficiency for ground data structure manipulation with Prolog-style logical variables by using PARMA bindings to ensure that the representation for terms used by HAL's Herbrand solver is consistent with that used by Mercury for ground terms. Prolog-like programs written in HAL run somewhat slower than in SICStus, in part because there is no term indexing for possibly unbound instantiations. However, once declarations are provided the programs run an order of magnitude faster. (Much of this arises from the sophisticated compilation techniques used by the underlying Mercury compiler.) Our results show that the biggest performance improvement arises from mode declarations while type and determinism declarations give moderate speed improvement. All declarations reduce the space requirements.

Acknowledgements

We would like to thank the Mercury development team, especially Fergus Henderson and Zoltan Somogyi, who have helped us with many modifications to the Mercury system to support HAL.

References

[1] H. Aït-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. In *Procs. of CP99*, to appear, 1999.

[3] Thomas Lindgren, Per Mildner, and Johan Bevemyr. On Taylor's scheme for unbound variables. Technical report, UPMAIL, October 1995.

[4] A. Mariën, G. Janssens, A. Mulkers, and M. Bruynooghe. The impact of abstract interpretation: an experiment in code generation. In *Proc. of the ICLP89*, pages 33–47, 1989.

[5] P. Van Roy. Can Logic Programming Execute as Fast as Imperative Programming? Report 90/600, UCB/CSD, Berkeley, California 94720, Dec 1990.

[6] 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.

[7] A. Taylor. PARMA–bridging the performance gap between imperative and logic programming. *Journal of Logic Programming*, 29(1–3), 1996.

[8] D. H. D. Warren. An abstract Prolog instruction set. Technical Report 309, SRI International, Menlo Park, U.S.A., Oct. 1983.