1 Herbrand Constraints in HAL

Bart Demoen[1], María García de la Banda[2], Warwick Harvey[2], Kim Marriott[2], David Overton[2], and Peter J. Stuckey[3]

[1] Department of Computer Science, Catholic University Leuven, Belgium
[2] School of Computer Science & Software Engineering, Monash University, Australia
[3] Department of Computer Science & Software Engineering, University of Melbourne, Australia

Abstract. Mercury is a logic programming language that is considerably faster than traditional Prolog implementations, but lacks support for full unification. HAL is a new constraint logic programming language specifically designed to support the construction of and experimentation with constraint solvers, and which compiles to Mercury. In this paper we describe the HAL Herbrand constraint solver and show how by using PARMA bindings, rather than the standard WAM representation, we can implement a solver that is compatible with Mercury's term representation. This allows HAL to make use of Mercury's more efficient procedures for handling ground terms, and thus achieve Mercury-like efficiency while supporting full unification. An important feature of HAL is its support for user-extensible dynamic scheduling since this facilitates the creation of propagation-based constraint solvers. We have therefore designed the HAL Herbrand constraint solver to support dynamic scheduling. We provide experiments to illustrate the efficiency of the resulting system, and systematically compare the effect of different declarations such as type, mode and determinism on the resulting code.

1.1 Introduction

The logic programming language Mercury [11] is considerably faster than traditional Prolog implementations for two main reasons. First, Mercury requires the programmer to provide type, mode and determinism declarations and information from these is used to generate efficient target code. Types allow a compact representation for terms, modes guide reordering of literals and multivariant specialization, and determinism is used to remove the overhead of unnecessary choice point creation. The second main reason for Mercury's efficiency is that variables can only be *ground* (i.e., bound to a ground term) or *new* (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, a technique that allows an execution to continue computation from a previous program state by logging information about prior states during forward computation and using it to restore the states again during backtracking. Trailing usually means

recording the state of unbound variables right before they become aliased or bound. Since Mercury's new variables have no run-time representation they do not need to be trailed.

This paper investigates whether it is possible to have Mercury-like efficiency, yet still support true logical variables. In order to do so we describe our experiences with HAL, a new constraint logic programming language that compiles to Mercury so as to leverage from Mercury's sophisticated compilation techniques. Like Mercury, HAL requires the programmer to provide type, mode and determinism declarations. Unlike Mercury, HAL was specifically designed to support the construction of and experimentation with constraint solvers [2].

In particular, HAL includes a built-in Herbrand constraint solver that provides full unification (without the occurs check), thus supporting logical variables. The Herbrand solver uses PARMA bindings [12] rather than the standard variable representation used in the WAM [1,14]. PARMA bindings represent equivalence of variable by keeping all equivalent variables in a cycle, as opposed to WAM bindings which implement a union-find style equivalence class. The use of PARMA bindings allows the solver to use essentially the same term representation for ground terms as does Mercury (see Section 1.4.4). 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]

An important feature of HAL is its use of type classes to distinguish between solver and non-solver types (i.e., types with an associated solver and types without) and for the hierarchical organisation of constraint solvers. Type classes allow a clean separation between a constraint solver's interface and its implementation, thus supporting experimentation with different solvers. We detail how HAL's Herbrand constraint solver fits into this hierarchy.

Another important feature of HAL is its support for user-extensible dynamic scheduling, that is intended to support communication between solvers and construction of efficient propagation-based solvers. We have therefore designed the HAL Herbrand constraint solver to support dynamic scheduling. Here we detail how this has been achieved with a PARMA-binding based solver. Again type classes allow us to distinguish between solvers that support dynamic scheduling and those that do not.

The HAL programmer may specify for a particular constructor type `t` whether `t` requires a Herbrand constraint solver (i.e. must support full unification) and, if so, whether this solver should support dynamic scheduling. The HAL compiler will then automatically generate an appropriate instance of the Herbrand solver for `t`. By requiring that constructor types that need a solver must be specified, HAL can take advantage of this to simplify the representation, analysis and compilation of constructors types that do not need a solver.

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

The results of our empirical evaluation of HAL and its Herbrand solver are very promising since they show that HAL is capable of using information from type, mode and determinism declarations as well as information about which types require true constraint solving and dynamic scheduling to significantly reduce the overhead of Herbrand constraint solving. In particular they show that, with appropriate declarations, HAL is almost as fast as Mercury (the extra overhead is mainly due to support for trailing), yet allows true logical variables. And while without declarations its efficiency is about half that of SICStus Prolog, with declarations it is an order of magnitude faster.

The experiments are also designed to systematically evaluate the effect of each kind of declaration (type, mode, determinism, need to support full-unification and dynamic scheduling) on the efficiency of HAL programs so as to determine where 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, while determinism declarations provide only moderate speedup. Also, although type declarations can also provide speedup, the use of polymorphic types can actually lead to slowdown. The overhead of unnecessary support for delay is noticeable but small.

The remainder of the chapter is organized as follows. In Section 1.2 we first introduce the HAL language by means of a simple example, and then examine the different declarations in some detail. Section 1.3 provides the general design of HAL's Herbrand solvers in terms of their interface and associated predicates, while Section 1.4 details their actual implementation. Next, we examine how dynamic scheduling is defined in HAL in Section 1.5 before detailing how we implement dynamic scheduling for Herbrand solvers in Section 1.6. We give our empirical evaluation in Section 1.7, discuss related work in Section 1.8, and conclude in Section 1.9.

1.2 The HAL Language

This section provides 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 Constraint Logic Programming (CLP) syntax, with variables, rules and predicates defined as usual (see, e.g., [10] for an introduction to CLP). The module system in HAL is similar to that of Mercury. A module is defined in a file, it `imports` the modules it uses and has `export` annotations on the declarations for the objects that it wishes to be visible to those importing it. Selective importation is also possible.

The core language supports integer, float, character, and string data types plus polymorphic constructor types (such as lists) based on these base types. However, this support is limited to assignment, testing for equality, and construction and deconstruction of ground terms. More sophisticated manipula-

tion is available by importing (or building) a constraint solver for each of the types involved.

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)

:- export typedef tower   -> (a ; b ; c).                           (L3)
:- export typedef pair(T) -> (T - T).                               (L4)
:- export typedef move    = pair(tower).                            (L5)
:- export typedef list(T) -> ([] ; [T|list(T)]) deriving herbrand.  (L6)

:- export pred hanoi(int,list(move)).                               (L7)
:-        mode hanoi(in ,no) is semidet.                            (L8)
hanoi(N,M) :- hanoi2(N,a,b,c,M-[]).                                 (L9)

:- pred hanoi2(int,tower,tower,tower,pair(list(move))).             (L10)
:- mode hanoi2(in ,in   ,in   ,in   ,oo) is semidet.                (L11)
hanoi2(N,A,B,C,M-Tail) :-
     ( N = 1 ->
          M = [A-C|Tail]
     ;    N > 1,
          N1 is N - 1,
          hanoi2(N1,A,C,B,M-Tail1),
          Tail1 = [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 constructor types used in and exported by this module. The type `tower` gives the names of the towers, `pair` defines a polymorphic pairing type, `move` defines a move as a pair of towers using a type equivalence, and `list` defines polymorphic lists. The type declaration for lists contains the directive `deriving herbrand` indicating to the HAL compiler to generate an instance of the Herbrand constraint solver for list types.

Line ($L7$) 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 ($L8$) is an example of a *mode of usage* declaration. The predicate `hanoi/2`'s first argument has mode `in` meaning that it will already be `ground` (i.e., bound to a ground term) when called, the second argument has mode `no` meaning that it will be `new` (i.e., never seen before) on calling and `old`

(i.e., possibly "constrained") on return.[2] 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` (here the mode `oo` means the argument is "constrained" on both call and return).

1.2.1 Declarations

As we can see from the above example, HAL allows programmers to annotate predicate definitions with type, mode, determinism declarations (modelled on those of Mercury). Like Mercury, it also provides purity declarations and type classes. Here we examine these issues in more detail.

Type declarations: Type declarations detail the representation format of a variable or argument. Types are defined using (polymorphic) regular tree type statements such as those shown in (*L3*)–(*L6*). As another example, the statement

```
:- typedef tree(K,I) -> (item(K,I) ; node(tree(K,I),K,tree(K,I))).
```

defines the type constructor `tree/2` for binary keyed tree types with key type K and item type I. The definition states that type constructor `tree/2` has two functors: `item/2`, which represents a leaf node and is used to store an item with its key, and `node/3`, which represents an internal binary tree node and is a used to store a key (for directing the search) and the two subtrees.

Equivalence types are also allowed. For example, the statement

```
:- typedef move = pair(tower).
```

defines the type constructor `move/0` as an equivalent name for type constructor `pair/1` with type constructor `tower/0` as argument. Note that the right-hand side is only allowed to contain type constructors not functors.

Ad-hoc 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 either integer subtraction or difference list pairing. Overloading is important in CLP languages since it allows the programmer to overload the standard arithmetic operators and relations (including equality) for different types, allowing a natural syntax in different constraint domains.

[2] We could have given the mode `out` which means that the list will be ground on return, but HAL's mode checker is not yet powerful enough to confirm this.

Mode declarations: Mode declarations specify how execution of a predicate modifies the "instantiation state" of its arguments. A mode is associated with each argument of a predicate and 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. Arguments of unknown structure (i.e., those associated with a variable type) can only have one of the *base* instantiation states: `new`, `old` or `ground`. We say that program variable X is `new` if it has not been seen by its associated constraint solver (if one exists), `old` if it has, and `ground` if X has a known fixed value.

The *base* modes are mappings from one base instantiation to another: we use two letter codes (`oo`, `no`, `og`, `gg`, `ng`) based on the first letter of the instantiation, e.g. `ng` is `new`→`ground`. The standard modes `in` and `out` are synonyms for `gg` and `ng`, respectively.

For terms with known structure, such as a list of moves, more complex instantiation states (lying between `old` and `ground`) may be used to describe the state. An example is

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

which defines an instantiation state in which the difference list pair is certainly constructed, but the elements in the pair may still be unbound variables. Note that the `bound` keyword may be dropped from the definition since this is HAL's default.

Fully understanding the above instantiation definition is more complex than it may first appear, since this requires combining the instantiation with the type. This is because the actual meaning of `old` for a program variable X depends on whether its constructor type `t` is a solver-type or not. If `t` is a solver type, it indicates that X might be possibly unbound. If it is not, X must be bound. This applies recursively to all types associated to the arguments of the term to which X is bound (if any). This allows the base instantiation `old` to be used as a shorthand for the most general instantiation state of an initialized (i.e., not `new`) program variable.

For example, in the instantiation `bound_difflist` the base instantiation `old` is used for variables with type `list(move)` (or, equivalently, `list(pair(tower))`). Thus, it is actually a shorthand for the instantiation

```
:- instdef old_list_of_move -> ifbound([] ; [old_move|old_list_of_move]).
:- instdef old_move -> bound(old_tower-old_tower).
:- instdef old_tower -> bound(a; b; c).
```

which indicates that a variable with instantiation `old_list_of_move` may be unbound (since it is enclosed by the `ifbound` keyword), but, if bound, it is either bound to an empty list or to a list with a bound move in the head,

and a tail with the same instantiation state. Note that `old` means `bound` for the `pair` and `tower` constructor types since they are not solver types. [3]

It is important to note that HAL does not allow nesting of the base instantiation `new` within a structure, i.e., all arguments in the structure must already be either ground or old. As we will see later, this ensures that all subparts of a data structure properly exist on the heap.

Instantiation declarations can be parametric in their instantiation variables. For example, the instantiation definition

```
:- instdef bound_list(I) -> bound([] ; [ I | bound_list(I) ]).
```

defines lists whose skeleton is fixed, and whose elements have instantiation `I`.

As we have seen, instantiations in HAL can be quite powerful. However, defining such instantiations can also be laborious, especially since they are often type specific. Fortunately, being able to use `old` as a shorthand for the most general instantiation state of any type as illustrated above, means the user rarely needs to define such instantiations.

Finally, modes can be defined using statements of the form $Inst_1 \rightarrow Inst_2$ where, as indicated before, $Inst_1$ describes the input instantiation state and $Inst_2$ describes the output instantiation state. Equivalence modes are also allowed. Examples are

```
:- modedef in(I) -> (I -> I).
:- modedef in = in(ground).
:- modedef out(I) -> (new -> I).
:- modedef out = out(ground).
:- modedef new2old_list_of_move = out(old_list_of_move).
```

Note that mode definitions can be parametric, i.e., contain instantiation variables such as `I` above. This is, however, not the case for predicate mode declarations which cannot contain variables. For more details about mode and instantiations in HAL the reader is referred to [4].

Determinism declarations: Determinism declarations detail how many answers a predicate may have. HAL uses the Mercury hierarchy: `nondet` means any number of solutions; `multi` at least one solution; `semidet` at most one solution; `det` exactly one solution. The determinism `erroneous` indicates a run-time error, while `failure` indicates the predicate always fails.

Type class declarations: HAL also provides type class and class instance declarations based on those of Mercury [7]. Type classes support *constrained*

[3] The `ifbound` form of instantiation definition is not available to the programmer, and is only generated internally by translation from `old`. This is because arbitrary `ifbound` instantiations are not checkable without sophisticated sharing analysis.

polymorphism by allowing the programmer to write code that relies on parametric types having certain associated predicates and functions. In particular, a `class` provides a name for a set of types (which are parameters to the type class) for which certain predicates and/or functions (called the *methods*) are defined, and which form its interface.

For example, one of the most important built-in type classes in HAL is

```
:- class eq(T) where [
        pred T = T,
        mode oo = oo is semidet ].
```

which defines types `T` that support equality testing, i.e., for which an implementation of the method `=/2` for mode of usage `oo = oo` exists. Note however that, like Mercury, all types in HAL have an associated "equality" for modes `in=out` and `out=in`, which correspond to assignment, construction or deconstruction, and which are implemented using specialised built-in procedures rather than implementation of the more general `=/2` method.

Instances of the `eq/1` class can be specified, for example, by the declaration

```
:- instance eq(pair(T)) <= eq(T) where [
           pred(=/2) is pair_1_SolveEqual ].
```

which declares the type `pair(T)` to be an instance of the `eq/1` type class, as long as `T` is also an instance of the class, and as long as there exists a predicate called `pair_1_SolveEqual` which appropriately implements the `=/2` method for type `pair(T)`. Most types support testing for equality, the main exception being for types with higher-order subtypes. Therefore, HAL automatically generates instances of `eq/1` (including the predicates implementing the `=/2` method) for all constructor types (such as `pair/1`) which do not contain higher-order subtypes and for which the programmer has not already declared an instance, thus removing this burden from the programmer.

One major motivation for providing type classes in HAL is that they provide a natural way of specifying a constraint solver's interface and allow us to naturally capture the notion of a type having an associated constraint solver: It is a type for which there is a method for initialising variables and a method for defining true equality. Thus, the built-in `solver/1` type class is defined by:

```
:- class solver(T) <= eq(T) where [
        pred init(T),
        mode init(no) is det ].
```

The above declaration indicates that the `solver/1` type class provides initialisation method `init/1`. The class definition also indicates that `solver/1` is a subclass of `eq/1` and, thus, any instance of `solver/1` must also be an instance of `eq/1`. Therefore, for type `T` to be in the `solver/1` type class, there must exist predicates implementing the methods `init/1` and `=/2` for

this type with mode and determinism as shown. The HAL compiler automatically inserts calls to `init/1` to initialize new variables and may generate calls to `=/2` because of normalization.

Purity declarations: Purity declarations [3] capture whether a predicate is `impure` (affects or is affected by the computation state), or `pure` (otherwise). By default predicates are pure. Any predicate that uses an impure predicate must have its predicate declaration annotated as either `impure` (so that it is also impure) or `trust pure` (so that even though it uses impure predicates it is considered pure). Calls to pure predicates can be reordered by the HAL compiler during mode analysis but predicate calls are never reordered past an impure predicate call.

Combined declarations: For predicates with only one mode, HAL, as Mercury, provides syntax for combining all declarations into a single line. For example, lines ($L7$) and ($L8$) in the `hanoi` example can be expressed as

```
:- export pred hanoi(int::in, list(move)::no) is semidet.
```

We will often use this compact form in the sequel.

1.3 Herbrand Constraint 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 supported by Mercury. If the programmer wishes to make use of more complex constraint solving for terms of some type `t`, then they must explicitly declare that they wish to use a Herbrand constraint solver for `t`.

This is achieved by adding the annotation `deriving herbrand` to the type definition. The HAL compiler will then automatically generate a Herbrand constraint solver for that constructor type. In order to do this, the compiler makes use of the following predicates and type classes defined in the system module:

```
:- export pred herbrand_init(T::no) is det.

:- class herbrand(T) <= solver(T) where [].

:- export impure pred var(T::oo) <= herbrand(T) is semidet.
:- export impure pred nonvar(T::oo) <= herbrand(T) is semidet.
:- export impure pred ===(T::oo,T::oo) <= herbrand(T) is semidet.
```

The first predicate implements the `init/1` method for any Herbrand type declared as instance of the `solver/1` class. The `herbrand/1` type class will

be used to identify the set of Herbrand types, i.e., the constructor types which support full unification (since every instance of `herbrand(T)` must also be an instance of `solver(T)`), and a number of non-logical operations commonly used in Prolog style programming such as `var/1`, `nonvar/1`, and `===/2`. The last three predicates implement such non-logical operations for any Herbrand type. Predicates `nonvar/1` and `var/1` can be used to test if a Herbrand variable is bound or not, respectively. Predicate `===/2` succeeds only if both arguments are identical unbound Herbrand variables.[4] Note that we could have included these predicates as methods in the `herbrand/1` class instead of simply adding the class constraint `herbrand(T)` to their predicate type declaration. However, since the implementation of such methods will be identical for all types in the class, that would only complicate matters.

As mentioned before, the HAL compiler automatically generates a Herbrand constraint solver for any constructor type annotated with `deriving herbrand`. In doing this the compiler generates appropriate instances for the `herbrand/1`, `solver/1` and `eq/1` classes. For example, in the `hanoi` module, since the types (`move`, `tower` and `pair`) are only manipulated when bound and, therefore, do not require the full power of unification, these types were not annotated with `deriving herbrand`. On the other hand, since the program uses difference lists, a Herbrand constraint solver is needed for the list type. Hence, the list type is defined as

```
:- typedef list(T) -> ([] ; [T | list(T)]) deriving herbrand.
```

The HAL compiler will then automatically generate the following declarations:

```
:- instance eq(list(T)) <= eq(T) where [
            pred(=/2) is list_1_SolveEqual ].

:- instance solver(list(T)) <= eq(T) where [
            pred(init/1) is system:herbrand_init ].

:- instance herbrand(list(T)) <= eq(T).
```

plus the definition of the predicate `list_1_SolveEqual` which implements unification specialised for the list data type as the general `=/2` method for lists. Exactly how this is done will be discussed in detail in the following section. Note that `herbrand_init/1`, implementing the `init/1` method, is already defined in the `system` module.

[4] `===/2` is analogous to Prolog `==/2` but only succeeds if both arguments are unbound variables. Determining if two non-variable arguments are identical in HAL would require recursively traversing and comparing the sub-terms in the arguments. Hence, every subtype of the term would require the ability to test equivalence. Simply testing if two variables are identical only depends on the topmost type constructor.

The reader might be wondering why there is a need for the programmer to distinguish types for which Herbrand solving is supported from those for which it is not, since one could have simply defined all constructor types as Herbrand types, provided full unification for them, and then relied on the compiler to replace calls to the Herbrand solver by more efficient calls to the term assignment, construction, etc, procedures provided by Mercury. The main reason to separate the types is one of efficiency. The problem is that the compiler is not always capable of detecting whether a more efficient procedure can be used since to do so requires examining reordering of literals. Another reason is that a slightly more compact representation can be used for non-Herbrand terms since there is no need to have a tag for the case where the term is a variable. Separating the types means that these overheads will always be avoided in the case of the far more common non-Herbrand types.

The above decision improves efficiency at the cost of code duplication. For example, since the type of lists with associated Herbrand solving support is different from that of lists without support, HAL needs to provide two library modules, one for each type. Furthermore, terms of one type cannot be unified with those of the other type.

1.4 Implementing Herbrand Constraint Solving

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

1.4.1 Term Representation and Manipulation in the WAM

The Warren Abstract Machine (WAM) [14,1] forms the basis of most modern Prolog implementations. Terms are stored on a heap,[5] which is an array of data cells. A cell is usually broken into two parts: a tag and a reference pointer. The most important tag values are REF (a variable reference), ATM (an atomic object, i.e., a non-variable term with arity 0), and STR (a structure, i.e., a non-variable term with one or more arguments). An unbound variable (on the heap) is represented by a cell with a REF tag and a pointer to itself. An atom is represented by a cell with tag ATM and a pointer into the atom table. 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 `f` and the arity n, and the next n cells hold the representations of t_1, ..., t_n. For example, a possible heap representation of the term $f(h(X), Y, a, Z)$ is shown in Figure 1.1.

[5] For simplicity, we ignore stack variables.

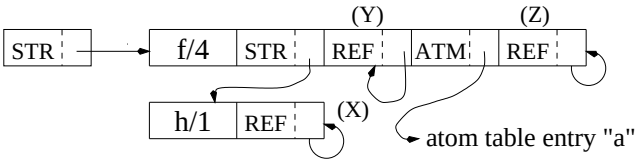
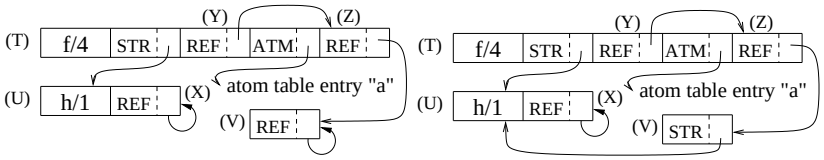


Fig. 1.1. WAM heap representation of $f(h(X), Y, a, Z)$.

The native representation of base types such as integers and floats (usually) uses the entire 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.

Unification of two objects on the heap proceeds as follows. 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) that object 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 equal, the corresponding arguments are unified.



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

Fig. 1.2. WAM term and variable binding schemes

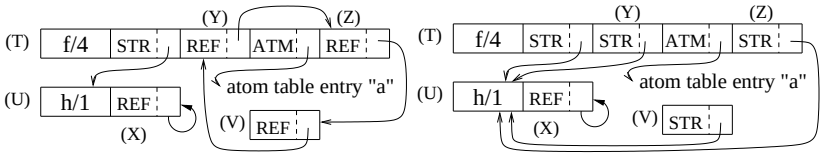
For example, consider the heap state of Figure 1.1. If we first unify Y with the heap variable Z and then with another heap variable V, we obtain the heap shown in Figure 1.2(a). If we then unify Y with $h(X)$ we obtain the heap shown in Figure 1.2(b). Notice how reference chains can exist throughout 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.

1.4.2 Term Representation and Manipulation in PARMA

In the PARMA system [12], 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 non-bound (i.e. 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 aliased are represented by n cells forming a cycle. When one of the variables is equated to a non-variable all variables in the cycle are changed to direct (tagged) pointers to this structure and changes are trailed.



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

Fig. 1.3. PARMA term and variable binding schemes

For example, the PARMA heap structures corresponding to Figures 1.2(a) and (b) are shown in Figures 1.3(a) and (b), respectively.

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. Furthermore, trailing of each variable requires two words (the variable's position and its old value).
(b) When instantiating a variable cycle (conditional) trailing must occur for each cell in the cycle (rather than one as for the WAM). Also, as before, the trail requires two words.
(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.

However, the impact of each of these factors is dependent on the length of the cycles that are manipulated. Since, as we shall see, cycles rarely grow beyond length one (a self pointer), the overhead involved is limited, although not completely eliminated (particularly in the case of trailing overhead).

It is important to 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.

1.4.3 Term Representation and Manipulation in Mercury

Types in HAL with no solver attached are identical to Mercury types. In this section we explain Mercury's approach to type representation and manipulation.

Recall that variables in Mercury can only be either `new` (which means they do not have a representation) or `ground`. Thus, there is no need for the REF tagged references used in the WAM. This combined with the fact that types are always known at compile time, allows Mercury to use a compact type-specific representation for terms in which tags are used instead to distinguish among the different type functors defined for the type. Hence, an object of a base type, like an integer, is free to use its entire cell to store its value. For more details see [11]. As an example, consider the Mercury type for lists:[6]

```
:- typedef list(T) -> ([] ; [T | list(T)] ).
```

Given a term of type `list(T)` 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 a representation of the first element (e.g. a tagged pointer or a 32 bit int) 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 four different tags are available. For 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 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 (although the arity of the functor does not need to be stored since the type information gives this). In what follows, we will ignore this for simplicity.

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.

As mentioned before, equations of the form $X = Y$ are only valid in three modes: `in = out`, `out = in`, and `in = in`. For the first two modes, the ground variable is copied into the new. For the third mode a procedure to check that the two terms are identical is called. Mercury automatically generates a specialized procedure (which we shall refer to as `unify_gg`) that does this for each type.

The equation $X = f(A_1, \ldots, A_n)$ is only valid in two modes: `out = in` (i.e., X is new and A_1, ..., A_n are all ground) and `in = out` (i.e., X is

[6] For uniformity we use HAL syntax rather than that of Mercury.

ground and each $A_1, \ldots, A_n$ is new). In the first case a contiguous block of n cells is allocated, the values of $A_1, \ldots, A_n$ are copied into these cells, and X is set to a pointer to this block with an appropriate tag. In the second case, after testing that X is bound to the appropriate type functor, the values in the contiguous block of n cells that it points to are copied into $A_1, \ldots, A_n$. The case where some of $A_1, \ldots, A_n$ are new and some ground (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 equations $X = 1, U = h(X), S = a, Z = Y, T = f(U, Y, S, Z)$. The first three equations can be compiled to "construct" variables X, U and S, respectively. 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 ground value by literal l, then these two equations can be reordered after l and compiled to construct Z and T.

1.4.4 Term Representation and Manipulation in HAL

Since HAL is compiled into Mercury, it makes considerable sense for HAL to use as far as possible Mercury's basic term manipulation functions even for types that sometimes require full unification. The idea is that, when possible, term equations should be compiled into Mercury's basic term manipulations (assignment, construction, deconstruction, and equality testing) rather than calling the more expensive unification solving method. However for this to be possible, terms in HAL must 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, once it is ground it becomes a legitimate Mercury term. Furthermore, 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 behaviour. In order to do this, HAL reserves the tag 0 in all Herbrand solver types for use as the REF tag. This means that instead of the four tags generally available for representing a type in Mercury there are only three available for a solver type.

For example, given the type declarations:

```
:- typedef erk -> (f(erk, erk, atm, erk) ; h(erk); g) deriving herbrand.
:- typedef atm -> (a ; b ; c ; d ; e).
```

the HAL representation of the term $T = f(h(X), Y, a, Z)$ is shown in Figure 1.4.

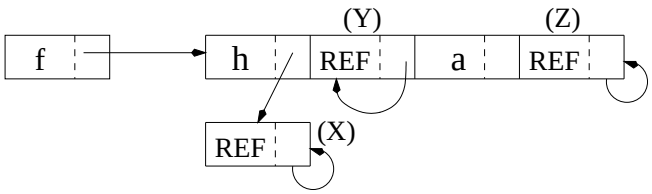


Fig. 1.4. HAL heap representation of $f(h(X), Y, a, Z)$.

Dereferencing: As in the PARMA system, only heap variables can be placed in a variable's alias cycle. Thus, a stack variable or a register must be a pointer somewhere into the cycle. As a result, when accessing data through a stack variable or register, HAL sometimes requires a single step dereference. Consider the following goal, where all variables are initially `new`:

```
init(Z), X = Z, X = [a], X = [A|B].
```

Figure 1.5 illustrates the changes to the heap and the registers holding `X` and `Z` during the execution of the first 3 atoms in the goal. Note that (due to the way Mercury handles registers) X and Z remain as pointers to the instantiated list rather than being updated to its value (what it points at on the heap). Before the execution of the atom `X = [A|B]` we must perform a one step dereference so that we can handle the equation simply as a Mercury deconstruct.

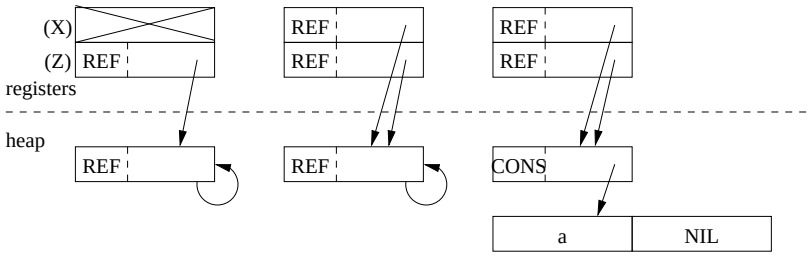


Fig. 1.5. Register and heap representation for each stage of `init(Z), X=Z, X=[a]`.

HAL produces Mercury code that maintains the assumption that:

- an `old` Herbrand object may need to be dereferenced.
- a `bound` Herbrand object is already dereferenced.

To do so, explicit dereferencing instructions are added to the output Mercury code, that create a new dereferenced version of a variable. Such dereferencing instructions are only required to be added to the user's code when the

compiler detects that the instantiation state of a variable changes from `old` to some bound instantiation. For example, the goal above is translated to Mercury code of the form

```
init(Z), X = Z, X = [a], X_Derefd = deref(X), X_Derefd = [A|B].
```

The `deref` pseudo-C code simply returns the value pointed to by its argument if this is not a variable[7]

```
deref(X) {
    if (derefd_var(X) && !derefd_var(*X)) return *X;
    return X; }
```

The code `derefd_var` to check whether a pointer is a variable pointer is simply

```
derefd_var(X) { return (tag(X) == REF); }
```

The code `var` to check whether an arbitrary `old` term is a variable must do the one step dereference. It is defined as follows:

```
var(X) { return (derefd_var(X) && derefd_var(*X)); }
```

The code for `nonvar` simply uses `var`.

```
nonvar(X) { return !var(X); }
```

Unification: HAL, as 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 as follows.

Consider an equation of the form $X = Y$. For modes `in = out`, `out = in`, and `in = in` we simply call the Mercury's more efficient procedures.[8] If one of the variables is new and the other one is old, we can simply assign the old variable to the new. This is identical to what Mercury does for this case (with the understanding that old is interpreted as ground) and we can therefore again use Mercury's procedure. 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. The (psuedo-C) code for `init` is simply

```
init(X) { X = top_of_heap++; *X = X; }
```

[7] Importantly the code does not return the next address in a variable chain, but the original address. This will be required later for correctness of dynamic scheduling.

[8] For `in = in`, this is correct only if X and Y contain no non-Herbrand solver types. For the purposes of this paper we will ignore this.

The only remaining case, where both X and Y are old, requires true unification. We replace the equation with a call to the Herbrand unification procedure `unify_oo`, which is automatically generated by the HAL compiler for the solver type `t` of X and Y.[9] A simplified version of the code for `unify_oo` is shown in Figure 1.6. In the actual code the calls to `nonvar` and `deref` are folded into one call.

```
:- pred unify_oo(T,T) <= herbrand(T).
:- mode unify_oo(oo,oo) is semidet.
unify_oo(X,Y) :-
     (nonvar(X) ->
          (nonvar(Y) ->
               unify_val_val(deref(X),deref(Y))
          ;    unify_var_val(Y,deref(X)))
     ;    (nonvar(Y) ->
               unify_var_val(X,deref(Y))
          ;    unify_var_var(X,Y))).
```

Fig. 1.6. HAL code for equating two `old` objects of type T.

The procedure `unify_val_val` is similar to Mercury's procedure `unify_gg` except it calls `unify_oo` on arguments of unified terms rather than `unify_gg`. It assumes that its arguments are dereferenced. For example, `unify_val_val` and `unify_gg` for list types are shown in Figure 1.7. In practice the final calls to `unify_oo` and `unify_gg` would be specialized since we know they apply to list arguments (and thus we know the name of the predicate which implements the method).

The procedure `unify_var_val` in Figure 1.8 unifies a variable and a non-variable. This means modifying all the variables in the cycle to directly refer to the non-variable, and trailing the changes. The procedure assumes the second argument is dereferenced.

The procedure `unify_var_var` shown in Figure 1.9 unifies two variables. This means checking that the variables are not already the same, and then joining the cycles together, trailing the change. Note that, unlike the case for the WAM, the code for unifying two variables is symmetric, treating each variable the same way. Also note that the algorithm traverses the two cycles in parallel stopping when the shortest cycle has been completed.

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. First, let us consider the simple case when X is bound, then the case when X is `new`, and finally the most complex case: when X is `old`.

[9] `unify_oo` is very similar to the code generated by the HAL compiler for the `=/2` method to ensure the type `t` is an instance of the `eq` class.

```
:- pred unify_gg(list(T),list(T)) <= eq(T).
:- mode unify_gg(in,in) is semidet.
unify_gg([],[]).
unify_gg([X|Xs], [Y|Ys]) :-
     unify_gg(X,Y),
     unify_gg(Xs,Ys).

:- instdef nonvar_list -> bound([]; [old|old]).
:- pred unify_val_val(list(T),list(T)) <= eq(T).
:- mode unify_val_val(in(nonvar_list),in(nonvar_list)) is semidet.
unify_val_val([],[]).
unify_val_val([X|Xs], [Y|Ys]) :-
     unify_oo(X,Y),
     unify_oo(Xs,Ys).
```

Fig. 1.7. HAL code for equating two nonvariable objects of type $list(T)$.

```
unify_var_val(X,Y) {
   QueryX = X;
   repeat
      {  Next = *QueryX;
         trail(QueryX);      /* trail chain pointer */
         *QueryX = Y;        /* replace by value */
         QueryX = Next; }
   until (QueryX == X) }
```

Fig. 1.8. Pseudo-C code for HAL unification of a variable and value

```
unify_var_var(X,Y) {
   QueryX = *X;
   QueryY = *Y;
   while (QueryX != Y && QueryY != X)    /* while equality not found */
      if (QueryX != X && QueryY != Y) {  /* if loops unfinished */
         QueryX = *QueryX;               /* advance */
         QueryY = *QueryY;
      } else {
         trail(X); trail(Y);             /* else trail X and Y */
         Tmp = *X; *X = *Y; *Y = Tmp;    /* merge chains */
         break; } }                      /* and finish */
```

Fig. 1.9. Pseudo-C code for HAL unification of two variables

The easiest 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 second case, when X is `new`, will require the construction of a new structure on the heap. For this to happen, and since arguments within a structure are not allowed to be new in HAL, each variable A_i with instantiation `new` must first be initialised. If the type of the variable is known at compile time to be a Herbrand type or other solver type, initialisation is not a problem. If, however, the type is known to be neither Herbrand nor any other solver-type, a compile-time error can be issued. Finally, if the type of the variable is not known at compile-time (i.e., it is a variable type), we must call a general initialisation procedure that decides what to call at run-time and can result in a run-time error if the type ends up not being a solver type. This would be simple if one could at run-time check whether a variable has a type which is an instance of certain type class (such as `herbrand/1` or `solver/1`). However, this is not yet possible in Mercury. Thus, in order to support this and other type-related queries, HAL defines the following internal type class:

```
:- class hal_type_info(T) where [
        pred maybe_init(T::no) is det,
        pred is_type_herbrand(T::oo) is semidet,
        pred is_type_solver(T::oo) is semidet].
```

where `maybe_init/1` initialises the variable in the heap if this is needed before performing a construction, `is_type_herbrand` succeeds if the type is Herbrand, and `is_type_solver` succeeds if the type is a non-Herbrand solver-type. HAL will also automatically create an instance of `hal_type_info/1` for every user-defined type `t` as follows. If `t` is neither Herbrand nor a solver type, the instance is:

```
:- instance hal_type_info(t) where [
           pred(maybe_init) is error,
           pred(is_type_herbrand) is fail,
           pred(is_type_solver) is fail].
```

where `error` will issue a run-time error, and `fail` will always fails. If `t` is not a Herbrand but a solver type, the instance is:

```
:- instance hal_type_info(t) where [
           pred(maybe_init) is init,
           pred(is_type_herbrand) is fail,
           pred(is_type_solver) is true].
```

where `init` is the predicate appearing in the `solver(t)` as the implementation of method `init/1`, `true` always succeeds and `fail` always fails. Finally, if `t` is a Herbrand type, the instance is:

```
:- instance hal_type_info(t) where [
           pred(maybe_init) is dummy_init,
           pred(is_type_herbrand) is true,
           pred(is_type_solver) is fail].
```

where `dummy_init` does nothing (as we will see, Herbrand variables do not require initialisation before a construction), and `true` and `fail` are as before.

Using the above predicates, the construction of term $X = f(A_1, \ldots, A_p)$ can be done as follows. Let us assume that all variables have variable type, variables $A_{o_1}, \ldots, A_{o_m}$ are `old` while $A_{n_1}, \ldots, A_{n_l}$ are `new`. Then, the translation to Mercury is essentially:

```
maybe_init(A_{n_1}), ..., maybe_init(A_{n_l}),
X = f(A_1, ..., A_p),
(is_type_herbrand(A_{n_1}) -> A_{n_1} = init_heap(X,n_1 - 1) ; true),
...,
(is_type_herbrand(A_{n_l}) -> A_{n_l} = init_heap(X,n_l - 1) ; true),
(is_type_herbrand(A_{o_1}) -> fix_copy(X,o_1 - 1) ; true),
...,
(is_type_herbrand(A_{o_m}) -> fix_copy(X,o_m - 1) ; true)
```

where the method `maybe_init` is first used to initialise all non-Herbrand `new` variables. Once this is done, the construction can be scheduled as a Mercury construct. Then, `is_type_herbrand` is used to perform a run-time check to see if the actual type of the arguments is a `herbrand` type and, if so, call specialised code to appropriately initialise the argument. This is done by the `init_heap`(X, i) function, which creates a self reference in the i^{th} slot of the heap region pointed to by X and returns it. Note that indices for slots on the heap start from 0 and, therefore, we must use `init_heap(X,`$n_j - 1$`)` rather than `init_heap(X,`n_j`)`. The function is defined as:

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

Note that `init_heap` is effectively a specialized version of `init/1` for the PARMA representation of variables inside data structures.

Finally, each `old herbrand` argument A_{o_k} was copied by Mercury into the new heap structure. For cases where this simple copy may not have achieved the desired result we need to call `fix_copy(`$X, o_k - 1$`)`. If A_{o_k} was an unbound variable, the copy performed by Mercury 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. Thus, `fix_copy` needs to add 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` must replace the contents of the o_k^{th} cell by what it refers to. The procedure is defined as:

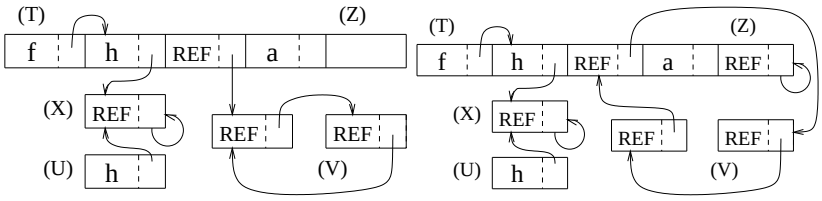
```
fix_copy(X,i) {
    AXi = &(X[i]); Xi = X[i];
    if (derefd_var(Xi))
        if (derefd_var(*Xi)) { trail(Xi); *AXi = *Xi; *Xi = AXi }
        else *AXi = *Xi; }
```

If, as it is usually the case, the types are known at compile time the generated code can be (and is) simplified enormously. Knowing the type allows the run-time type checks to be eliminated and the code simplified appropriately.

For example, consider the construction of $T = f(U, V, S, Z)$ where T and Z are new, 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). In this case we know the type of all arguments completely. The generated code is

```
maybe_init(Z),      %% Noop as Z is Herbrand
T = f(U,V,S,Z),     %% Mercury construct
Z = init_heap(T,3),%% initialize Z
fix_copy(T,1)       %% fix V
```

After executing the Mercury construction $T = f(U, V, S, Z)$ the heap is as shown in Figure 1.10(a). Applying `init_heap(T,3)` and `fix_copy(T,1)` gives the heap shown in Figure 1.10(b).



(a) After Mercury construct (b) Corrected version

Fig. 1.10. Adapting Mercury's term construction for Herbrand terms

To illustrate polymorphic code, consider the literal $X = [A|Y]$ where both X and Y have type `list(T)`, A has type `T`, X is `new` and both A and Y are `old`. The construction code is shown below:

```
X = [A|Y]                       %% Mercury construct
(is_type_herbrand(A) ->         %% if A is a term solver type
     fix_copy(X,0) ; true),     %% fix A
fix_copy(X,1)                   %% fix Y
```

The third and final case handles the equation $X = f(A_1, \ldots, A_p)$ 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[10] and then equates X to this term using `unify_var_val`.

Consider again the literal $X = [A|Y]$ where both X and Y have type `list(T)` and A has type `T`, this time with A `new` and both X and Y `old`. The generated code has the form

[10] Depending on whether arguments are solver types or not this may not be possible, causing a run-time error.

```
(nonvar(X) ->                              %% deconstruct
     Xd = deref(X),
     Xd = [An|Yn],                         %% Mercury deconstruct
     A = An,                               %% copy operation (A is new)
     unify_oo(Y,Yn)                        %% arbitrary unification
;                                          %% construct
     maybe_init(A),                        %% possible initialization of A
     X = [A|Y],                            %% Mercury construct
     (is_type_herbrand(A) ->               %% if A is a term solver type
          A = init_heap(X,0) ; true),      %% fix A
     fix_copy(X,1))                        %% fix Y
```

Again a run-time error can occur if X is a variable, since the call to `maybe_init` will raise an exception if A does not have a solver type.

1.4.5 Implementation of `herbrand/1` Methods

Supporting the methods in the `herbrand` type class is straightforward once the representation of terms is decided. We have already defined `var/1` and `nonvar/1` in Section 1.4.4. The `===/2` predicate only needs to check whether two variables are in the same reference chain. This can be implemented as follows (cf. the code for unifying two variables in Figure 1.9).

```
===(X,Y) {
   if (!var(X) || !var(Y)) return FALSE; /* not both vars */
   QueryX = *X; QueryY = *Y;
   while (QueryX != Y && QueryY != X)    /* while equality not found */
      if (QueryX != X && QueryY != Y) {  /* if neither loop finished */
         QueryX = *QueryX;               /* advance */
         QueryY = *QueryY;
      } else
         return FALSE;                   /* not identical */
   return TRUE; }
```

1.5 Dynamic Scheduling

Most modern logic programming languages allow predicates or goals to delay until a particular condition (such as becoming bound or being unified with another variable) is satisfied. Essentially they are implemented by hooks in the unification algorithm using attributed variables [6]. SICStus Prolog provides the ability to suspend a goal until a term is instantiated, ground or two terms are either identical or definitely not identical, and conjunctions and disjunctions of these. ECL^{i}PS^e provides the ability to suspend a goal until a term is bound to a variable or instantiated, and provides a user extensible hook (constrained) which is used to indicate any change made to a variable by a

constraint solver. In HAL, dynamic scheduling hooks (we call them delay conditions) are implemented by individual constraint solvers, and are completely extensible.

In the remainder of this section we describe the general dynamic scheduling mechanisms of HAL, and how Herbrand solvers fit into this scheme. In the next section we discuss how this is implemented.

1.5.1 Dynamic Scheduling in HAL

The HAL language provides a form of more "persistent" dynamic scheduling designed specifically to support constraint solving. A delay construct is of the form

$$cond_1 \texttt{ ==> } goal_1 \texttt{ | } \cdots \texttt{ | } cond_n \texttt{ ==> } goal_n$$

where the goal $goal_i$ will be executed *every time* the delay condition $cond_i$ is satisfied. This is useful, for example, if the delay condition is satisfied every time the lower bound of a solver variable has changed. Delayed goals may also contain calls to the special predicate `kill/0`. When this is executed, all delayed goals in the immediate surrounding delay construct are killed; that is, will never be executed again.

The delay construct of HAL is designed to be extensible, so that programmers can build constraint solvers that support delay. In order to do so, one must create an instance of the `delay` type class defined as follows:

```
:- class delay(D,I) <= delay_id(I) where [
        pred delay(D, I, pred),
        mode delay(oo, in, in(pred is semidet)) is semidet ].
:- class delay_id(I) where [
        impure pred get_id(I::out) is det,
        impure pred kill(I::in) is det ].
```

where type `I` represents the unique identifier (id) of each delay construct, type `D` represents the supported delay conditions (such as `bound(X)` in the case of the Herbrand solver), `delay/3` takes a delay condition, an id and a goal,[11] and stores the information in order to execute the goal whenever the delay condition holds, `get_id/1` returns an unused id, and `kill/1` causes all goals delayed for the input id to no longer wake up.

The HAL compiler translates each delay construct into the base delay methods provided by the classes as follows. Consider the generic delay construct shown above. This construct is translated into:

`get_id(Id), delay(`$cond_1$`,Id,`$goal'_1$`), ..., delay(`$cond_n$`,Id,`$goal'_n$`)`

[11] To simplify analysis, each $goal_i$ must be `semidet` and may not change the instantiation state of variables. As a result, the possibility of delayed code waking up can be ignored during mode and determinism checking since such code can never change the current instantiation or determinacy.

where each call to `kill/0` in $goal_i$ is replaced by a call to `kill(Id)` in $goal'_i$. The separation of the delay type class into two parts allows different solver types to share delay ids. Thus, we can build delay constructs which involve conditions belonging to more than one solver as long as they use a common delay id.

As mentioned above, a constraint solver supporting dynamic scheduling must declare an instance of the `delay/2` type class. In order to do so it needs to

- define a type `D` expressing the kinds of allowable delay conditions;
- define a type `I` for representing identities (ids) for delay constructs;
- define the predicate `get_id/1` which returns a new unused delay id;
- define the predicate `kill/1` which causes all delaying code with the input delay id to no longer wake up (and hence effectively be removed from the solver); and
- define the predicate `delay/3` which takes a delay condition, delay id and a goal, and stores the information in order to execute the goal when the delay condition holds.

If the programmer uses the annotation `deriving delay` instead of using `deriving herbrand` when defining a constructor type `t`, the compiler will automatically generate a Herbrand constraint solver for `t` that supports delay. As we will see later, the reason to distinguish between Herbrand solvers that support delay and those which do not is a matter of efficiency: the implementation of delay for Herbrand solvers introduces an overhead that HAL programmers might wish to avoid when support for dynamic scheduling is not needed.

In order to generate a Herbrand solver that supports delay, the HAL compiler makes use of the following types, classes, instances and predicates defined in the system module:

```
:- export_abstract typedef herbrand_delay_id = int.
:- export typedef delay_cond(T) -> (bound(T) ; touched(T)).

:- export class herbrand_delay(T) <= herbrand(T) where [].
:- export instance delay_id(herbrand_delay_id).
:- export instance delay(delay_cond(T),herbrand_delay_id) <=
                        herbrand_delay(T).

:- export impure pred get_id(herbrand_delay_id).
:-               mode get_id(out) is det.

:- export impure pred kill(herbrand_delay_id).
:-               mode kill(in) is det.

:- export pred delay(delay_cond(T),herbrand_delay_id, pred) <=
                    herbrand_delay(T).
```

```
:-        mode delay(oo, in, in(pred is semidet)) is semidet.
```

The module defines the type `herbrand_delay_id` as an integer and abstractly exports it (i.e. the type is visible from outside but its particular definition is not). It also exports the type `delay_cond(T)` which defines the delay conditions supported for a herbrand variable of type `T`: `bound(X)` will succeed whenever variable `X` becomes bound, while `touched(X)` will succeed whenever variable `X` becomes bound or aliased to another variable *which also has associated delayed goals*. While the `bound(X)` condition will succeed at most once, the `touched(X)` condition may succeed more than once. Note that `touched(X)` does not wake when X is bound to a variable without any associated delayed goals since such a unification does not change the "meaning" of the constraint store.[12]

The purpose of the `herbrand_delay/1` class is simply to record which Herbrand types support delay. The rest of the module exports the instances of classes `delay_id/1` and `delay/2` which will be used by all Herbrand constraint solvers that support delay, and the predicates which implement the associated methods. All Herbrand solvers which support delay will use the common delay conditions `bound(X)` and `touched(X)`, the common delay id type `herbrand_delay_id`, and its system-defined instance of `delay_id`. Note, however, that `herbrand_delay_id` can also be used by user-defined solvers.

Based on the above types and classes, the only difference at compile-time between a type defined as `deriving herbrand` and one defined as `deriving delay` is that, for the latter, the HAL compiler automatically generates an instance of the `herbrand_delay/1` class, in addition to those of `herbrand/1`, `solver/1`, and `eq/1` classes which are generated for both types.

As an example of the use of delay, the following code shows (part of) a simple Boolean constraint solver which is implemented using Herbrand constraint solving.

```
:- export typedef boolv -> ( f ; t ) deriving delay.
:- export pred and(boolv::oo,boolv::oo,boolv::oo) is semidet.
and(X,Y,Z) :-
  ( bound(X) ==> kill, (X = f -> Z = f ; Y = Z)
  | bound(Y) ==> kill, (Y = f -> Z = f ; X = Z)
  | bound(Z) ==> kill, (Z = t -> X = t, Y = t ; notboth(X,Y))).
:- export trust pure pred notboth(boolv::oo,boolv::oo) is semidet.
notboth(X,Y) :-
  ( bound(X) ==> kill, (X = t -> Y = f ; true)
  | bound(Y) ==> kill, (Y = t -> X = f ; true)
  | touched(X) ==> (X === Y -> kill, X = f ; true)
  | touched(Y) ==> (X === Y -> kill, X = f ; true)).
```

[12] This is analogous to the case of unifying an attributed variable to a non-attributed variable.

The constructor type `boolv` is used to represent Booleans. Since the type is defined as `deriving delay`, the compiler will automatically generate instances of the classes `herbrand_delay/1`, `herbrand/1`, `solver/1` and `eq/1`. Thus `old` variables of this type are allowed and represent unknown Boolean values.

The Boolean constraint solver defines two constraints: `and(X,Y,Z)` which implements the formula $X \wedge Y \leftrightarrow Z$, and `notboth(X,Y)`, which implements the formula $\neg X \vee \neg Y$. Both constraints are defined using dynamic scheduled code. The code for `and(X,Y,Z)` delays until one of its arguments is bound (which for this type is equivalent to ground), and then executes once (it is immediately killed on wake up). If either X or Y is bound the constraint is solved. If Z is bound to `f` the constraint `notboth(X,Y)` is created. Note that we could also have made use of `touched` delay conditions in the definition of `and`.

The code for `notboth(X,Y)` delays until either X or Y is bound in which case the constraint is enforced, or if X or Y is touched (bound or unified with a different variable which also has delayed code). In the second case if X and Y are identical (`===`), the delay construct is killed and both are set to *false* (the only way to satisfy the constraint), otherwise the construct remains. This illustrates how delayed code can be executed multiple times. Note that `notboth/2` uses the `impure` predicate "`===`," however, since the actions of `notboth` as seen from the outside are pure, we use a `trust pure` declaration for the constraint.

To illustrate how dynamic scheduling works, consider the execution of goal:

```
and(A,B,C), and(A,C,D), and(A,E,F), D = f, C = G, A = E, B = t.
```

where all variables are assumed to have just been initialised. Initially all three `and` constraints delay. When the constraint `D = f` is executed, `and(A,C,D)` wakes up, kills its delay construct and calls `notboth(A,C)` which delays. When `C = G` is executed, no delayed goal wakes up since there is nothing delaying on `G`. When `A = E` is executed, `notboth(A,C)` wakes (since A is touched) but since `A === C` fails the wake up does nothing. Executing `B = t` wakes `and(A,B,C)`, kills its delay construct and adds the constraint `A = C`. This wakes `notboth(A,C)` since it causes a `touched` event on A (and C), finds that they are identical, kills its delay construct and sets both A (and C through the equality) to `f`. This wakes `and(A,E,F)` which kills its delay construct and sets F to `f`. The solution gives $A = C = D = E = F = G =$ `f` and $B =$ `t`.

Currently HAL only supports simple delay conditions, rather than conjunctions or disjunctions of delay conditions. For example, it would be convenient to replace the last two lines of constraint `notboth(X,Y)` by the single line

```
(touched(X);touched(Y)) ==> (X === Y -> kill, X = f ; true)
```

These more complex delay conditions are not directly supported by HAL yet, but can be implemented by straightforward program transformation.

1.6 Implementing Dynamic Scheduling

In this section we begin by discussing the usual approach to implementing dynamic scheduling for Herbrand constraints in the WAM, then we consider how it is implemented in HAL.

1.6.1 Implementing Dynamic Scheduling in the WAM

Most Prolog systems, including SICStus Prolog and ECL^{i}PS^e support dynamic scheduling based on Herbrand constraint solving using attributed variables [6]. For simplicity we shall illustrate the delay mechanism assuming a single (delay) attribute, and only explain waking up when a variable is bound to a non-variable using the builtin `freeze` which corresponds to the delay condition `bound`. See also the section on Attributed Variables in [5] for a more detailed explanation.

Essentially a new kind of variable is introduced, which we will represent using the tag ATT. An attributed variable is stored in two contiguous data cells. The first cell acts like a variable, while the second cell is where we store the attributes of the variable, which for our purposes is a list of goals to be executed when the variable is bound to a non-variable.

The goal `freeze(X,G)` creates a new attributed variable `Y` with attribute `[G]`, and then unifies it with `X`.

Unification is extended to attributed variables as follows. When an attributed variable `X` is unified with a non-variable term, then all the delayed goals in the delay attribute of `X` are executed. If an attributed variable `X` is unified with another attributed variable `Y`, then the two lists of delayed goals are concatenated, and this replaces the delayed goal for `Y` (say), and `X` is pointed at `Y`.

Consider the goal

```
G = write(X), freeze(X,G), H = write(g(Y)), freeze(Y,H), X = Y, X = f(Z).
```

then after the first four literals executes the heap holds the two attributed variables `X` and `Y` with their delayed goals. The heap state is shown on the left of Figure 1.11. On the unification of `X` and `Y` the two lists are appended and the attribute of `Y`, and `X` is pointed at `Y`, resulting in the heap state in the middle of Figure 1.11. When `X` is bound to `f(Z)` it is first dereferenced to obtain `Y`, the goal list `[G,H]` is remembered for execution, and `Y` pointed to `f(Z)`. The heap state is now as in the right of Figure 1.11. The delayed goals are then executed, causing `f(Z)g(f(Z))` to be printed (although the other order `g(f(Z))f(Z)` is equally probable in practice).

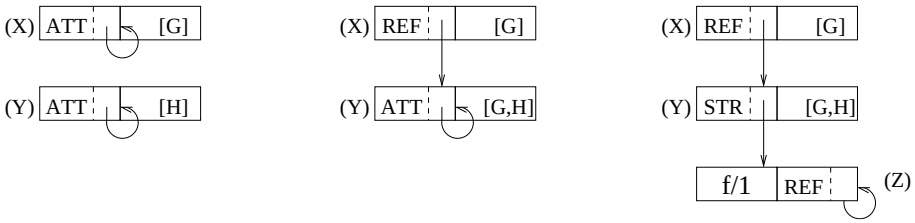


Fig. 1.11. WAM heap representation for dynamically scheduled goals and after executing each literal `X = Y, X = f(Z)`.

Prolog systems typically include a global register for holding all the delayed goals scheduled. The goals in this register are executed only at certain points in the code, typically just before a predicate call is made.

1.6.2 Implementing Dynamic Scheduling in HAL

As we saw in Section 1.5.1, each delay construct is converted by the compiler to a more low-level set of delay primitives: `get_id/1`, `kill/1` and `delay/3`. In the following subsections we will explain how the procedures `get_id/1`, `kill/1` and `delay/3` are implemented for Herbrand solvers.

1.6.3 Storing Dynamically Scheduled Goals

Herbrand delay conditions `bound(X)` and `touched(X)` are associated with variable `X` by placing an entry in the alias cycle associated with `X`. Since each entry in the alias cycle must be a variable, they all have a variable tag (REF). Thus, we can use any other tag (which is already used by the type) to represent a delay node (DEL). We use tag 1.

A delay node is stored as four consecutive heap cells as shown in Figure 1.12. These four components are: a dummy variable node which points to the next component, the DEL tagged delay node pointing to the next variable in the alias chain, a pointer to the doubly linked list of goals to be woken on a bound event, and a pointer to the doubly linked list of goals to be woken on a touched event. The system maintains at most one delay node in any alias cycle. The apparently unnecessary extra (dummy) variable node allows us to ensure that we never encounter the DEL tagged node in a context where it might be confused with the usual functor that uses tag 1. In particular, `fix_copy` performs a one step dereference on things which appear to be variables; we need to make sure it doesn't encounter a delay node at that point or it will mistake it for a bound term. Note that this also means that we should take care when dereferencing a variable, since if we store the resultant address we may have a direct pointer to the dummy node, which if dereferenced will incorrectly appear to be a bound term.

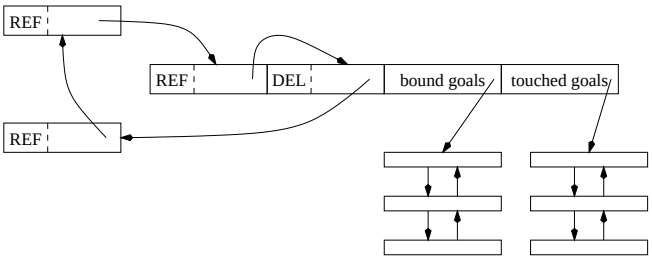


Fig. 1.12. A delay node within an alias cycle.

Adding a dynamically scheduled goal to the alias cycle is straightforward. We search the alias cycle for a delay node; if there isn't one we create a new empty one and place it in the cycle. We then add the goal to the appropriate doubly linked list of goals (depending on the delay condition). Note that if the variable is already bound, then the goal is simply executed immediately.

1.6.4 Modifying Unification for Delay

For `herbrand_delay` types we need to modify the code for manipulating variables in order to recognize when a delay condition has been satisfied. When unifying an alias cycle with a structure we know that both `bound(X)` and `touched(X)` for any variable `X` in the chain is satisfied. Thus, we need to adjust the `unify_var_val/2` algorithm to detect whether a delay node appears in the chain and, if so, execute both lists of delayed goals. The code is shown in Figure 1.13 (cf. the original code in Figure 1.8). If we detect that the next item in the chain has a DEL tag then we are currently looking at the dummy variable in the chain, and the next element is the delay node. We record this and skip past the delay node. Otherwise we proceed as usual. If after traversing the chain we have detected a delay node, we execute both lists of delayed goals.

Unifying two alias cycles is more complex, as shown in Figure 1.14. If only one variable chain contains a delay node we proceed as in Figure 1.9. If both contain a delay node then we need to merge their delay nodes, and also wake up goals with a touched delay condition. Note that we have to be careful not to insert an extra node in between the two cycle elements in a delay node.

If the variables are the same we immediately return, otherwise we look through the X cycle until we either find Y (in which case we return), or find a delay node, or complete the cycle. We then look through the Y cycle until we either find X, in which case we return, or find a delay node or complete the cycle. If we found no delay nodes we proceed as before. If we find one delay node, we insert the other chain just after the delay node. If we find two delay nodes we merge the lists of delayed goals into the delay node for

```
unify_var_val(X,Y) {
   QueryX = X;
   DelayNode = null;
   repeat {
      Next = *QueryX;
      if (tag(*Next) != REF) {              /* Found delay node */
         DelayNode = Next;                  /* save in DelayNode */
         QueryX = (strip_tag(*Next));       /* continue */
      } else {
         trail(QueryX);
         *QueryX = Y;
         QueryX = Next;
      } }
   until (QueryX == X)
   if (DelayedNode) {
      execute_delayed_goals(*(DelayNode+1)); /* execute bound goals */
      execute_delayed_goals(*(DelayNode+2)); /* execute touched goals */
   } }
```

Fig. 1.13. Pseudo-C code for HAL unification of a variable and value supporting delay

X (using `merge_delay_goals`) and then insert the the X cycle just after the dummy node in the cycle of Y, stripping out the rest of the delay node.[13]

We now illustrate the execution of the same goal, as previously considered for the usual Prolog approach

```
G = write(X), freeze(X,G), H = write(g(Y)), freeze(Y,H), X = Y, X = f(Z).
freeze(X,G) :- (bound(X) ==> call(G)).
```

then after the first four literals executes the heap holds the two attributed variables `X` and `Y` and their delay nodes holding the delayed goals. The heap state is shown on the top of Figure 1.15. On the unification of `X` and `Y` the two lists are appended and the cycles are merged, eliminating the delay node of `Y`, resulting in the heap state in the middle of Figure 1.15. When `X` is bound to `f(Z)` the goal list `[G,H]` is remembered for execution, and every (non-delay) element in the cycle for `X` is pointed to `f(Z)`. The heap state is now as in the bottom of Figure 1.15. The delayed goals are then executed, causing `f(Z)g(f(Z))` to be printed.

As we can see the heap usage by the HAL representation is more complicated than the corresponding WAM representation. Note also that the addition of delay for a solver type potentially slows down all unifications for that type since we may need to search both alias cycles to determine if

[13] Actually by keeping track of the previous pointers we can avoid using the dummy node for Y, unless the delay nodes are the first things we encounter in both chains.

```
unify_var_var(X,Y) {
  if (X == Y) return;                   /* shortcut return */
  QueryX = X;
  DelayNodeX = null;
  repeat {                              /* search for delay node in X */
    NextX = *QueryX;
    if (NextX == Y) return;             /* shortcut return */
    if (tag(*NextX) != REF) {           /* found delay node */
      DelayNodeX = NextX;
      break; }
    QueryX = NextX; }
  until (QueryX == X);
  if (DelayNodeX == null) {             /* no delay in X, just unify */
    NextY = *Y;                         /* search for insert place */
    if (tag(*NextY) != REF) {           /* found delay node */
      DelayNodeY = NextY;
      trail(X); trail(DelayNodeY);      /* add X to cycle for Y */
      Tmp = strip_tag(*DelayNodeY);     /* after Ys delay node */
      *DelayNodeY = add_tag(DEL,*X);
      *X = Tmp;
    } else {                            /* otherwise Y not dummy node */
      trail(X); trail(Y);
      Tmp = *X; *X = *Y; *Y = Tmp; }
    return; }
  QueryY = Y;
  DelayNodeY = null;
  repeat {                              /* search for delay node in Y */
    NextY = *QueryY;
    if (NextY == X) return;             /* shortcut return */
    if (tag(*NextY) != REF) {           /* found delay node */
      DelayNodeY = NextY;
      break; }
    QueryY = NextY; }
  until (QueryY == Y);
  if (DelayNodeY == null) {             /* add Y to cycle for X */
    trail(Y); trail(DelayNodeX);        /* after Xs delay node */
    Tmp = strip_tag(*DelayNodeX);
    *DelayNodeX = add_tag(DEL,*Y);
    *Y = Tmp;
  } else if (DelayNodeY == DelayNodeX) /* same variable */
    return;
  else {
    merge_delay_goals(DelayNodeX, DelayNodeY); /* merge into X delay */
    trail(QueryY); trail(DelayNodeX);
    *QueryY = strip_tag(*DelayNodeX);
    *DelayNodeX = *DelayNodeY;
    execute_delayed_goals(*(DelayNodeX+2)); /* execute touched goals */
  } }
```

Fig. 1.14. Pseudo-C code for HAL unification of two variables supporting delay

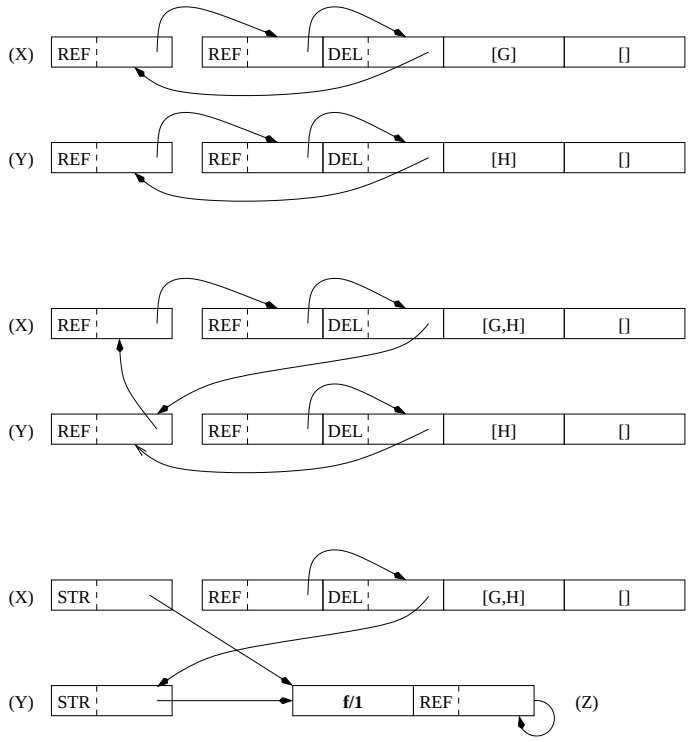


Fig. 1.15. HAL heap representation for dynamically scheduled goals and after executing each literal `X = Y, X = f(Z)`.

we have delay nodes in them. That is why HAL requires the user to explicitly indicate whether a Herbrand type requires support for delay, so that it can generate calls to the more efficient versions of `unify_var_val` and `unify_var_var` where possible.

1.6.5 Killing Dynamically Scheduled Code

Because the dynamically scheduled code is potentially executed multiple times, the delay constructs need to be explicitly killed when they are no longer needed. As we have seen before, for Herbrand constructs the `herband_delay_id` type is an integer and the `get_id` predicate is thus implemented using a global integer counter. The ability to kill dynamically scheduled code is managed by associating with each `herband_delay_id` the list of delayed goal nodes that make up the construct. The `kill/1` predicate simply traverses this list removing each delayed goal node from the doubly linked list in which it occurs.

1.7 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[14] and compare them with the equivalent HAL programs. In order to build these equivalent programs we must first transform built-ins not present in HAL (such as cut) into their HAL equivalents (such as if-then-else). Also, although Prolog does not have type, mode and determinism declarations, the current HAL compiler requires them. We solve this problem by defining a "universal" constructor type for the HAL program which contains all functors occurring in the program and declaring this type to be a Herbrand solver type supporting dynamic scheduling by using `deriving delay`.

Note that all integers, floats, chars and strings in the original Prolog program must be wrapped in the HAL program, and each wrapping functor must appear in the "universal" constructor type. Finally, all predicate arguments are declared to have this type and mode `oo`, and all predicates are declared to have determinism `nondet`. Most of these tasks are done automatically by a pre-processor.

For example, for the original `hanoi` Prolog program (the code in Section 1.2 minus the declarations), the preprocessor will add the declarations

```
:- typedef htype -> (int(int) ; float(float) ; a ; b ; c ; []
                    ; [htype|htype] ; mv(htype,htype) ; htype-htype )
                    deriving delay.
:- 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.
```

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

To achieve our second aim of investigating the impact of each kind of declaration on efficiency, we take these Prolog-equivalent HAL programs and progressively transform them as follows.

- The first step is to add precise type information, i.e., to add the required type definitions and acccurate predicate type declarations. All types must still be declared as Herbrand solver types supporting delay since the associated terms may sometimes be treated as logical variables. This also

[14] See `http://www.csse.monash.edu.au/~mbanda/hal`.

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.
- The second step is to remove the support for dynamic scheduling for those Herbrand solver types upon which nothing is delayed. We simply replace the directive `deriving delay` by the directive `deriving herbrand` wherever possible.
- The third step adds accurate mode declarations. Types which are never associated with the `old` instantiation need not be declared as Herbrand solver types (i.e. their `deriving herbrand` directive is removed) and, in the case of the primitive types, such types can have their wrapping removed.
- In the fourth and last step precise determinism declarations are added.

We then evaluate the efficiency of the programs obtained at each step.

Our third and final aim is to compare the efficiency of HAL and Mercury to determine the overhead introduced by the run-time support for HAL, i.e., the overhead introduced by trailing, the reserved REF tag used for solver-types, extra type classes, predicate renamings, etc. In order to do so we took the program resulting from compiling the HAL program obtained in the fourth step above, and modified it by using the Mercury libraries (instead of HAL ones), eliminating any unification-related code (which was actually dead-code anyway), and eliminating any predicate renaming introduced due to the use of type classes, etc. The resulting program was then compiled using two different compilation grades of Mercury: one that does not provide trailing and one that does. Both grades also avoid reserving the extra REF tag for solver-types, but are otherwise equivalent to the Mercury grade used for compiling the HAL programs. Note that since Mercury does not provide full unification, we could only do this for benchmarks with no remaining herbrand types.

All timings are in seconds on a dual Pentium II-400MHz with 632M of RAM running Linux 2.2.9. We have turned garbage collection off in all three systems: SICStus Prolog 3.8.6 (compact code), Mercury (release-of-the-day 2003-08-09 version), and HAL.

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[15] and hence the comparison is fairer. To this we added two HAL benchmarks using

[15] `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.

delay, both based around Boolean constraint solving. The first `bqueens` is the classic n-queens problem, the second `nono` is a nonogram solver.[16]

Benchmark	Preds	Lits	OSICS	SICS	None	T	TS	TSM	TSMD	Merc+tr	Merc
aiakl	7	21	0.09	0.08	0.39	0.94	0.97	0.02	0.03	0.03	0.01
boyer	14	124	1.79	0.51	2.36	2.00	2.23	0.11	0.05	0.08	0.02
bqueens	23	99	—	73.38	4.86	5.04	5.04	4.77	4.73	—	—
deriv	1	33	1.54	2.41	5.02	4.88	4.08	0.83	0.68	0.69	0.15
fib	1	6	1.20	1.21	0.36	0.33	0.27	0.02	0.02	0.01	0.01
hanoiapp	2	7	2.57	2.61	6.30	14.36	13.77	0.64	0.32	0.27	0.19
hanoidiff	2	6	1.81	1.75	0.54	0.73	0.74	0.66	0.63	—	—
mmatrix	3	7	1.26	1.26	1.22	2.96	2.35	0.10	0.05	0.04	0.01
nono	30	181	—	16.35	11.21	17.56	17.56	2.12	2.08	—	—
qsortapp	3	10	2.94	1.60	5.14	10.13	10.10	0.51	0.22	0.21	0.11
qsortdiff	3	10	2.91	1.64	5.22	9.92	10.06	0.53	0.24	—	—
serialize	5	19	1.41	1.36	2.30	2.56	2.83	0.63	0.46	—	—
tak	1	9	0.49	0.60	0.90	0.76	0.68	0.08	0.06	0.05	0.01
warplan	25	88	0.51	0.60	2.12	1.14	1.06	0.40	0.32	—	—
Average				1.16	0.77	0.77	1.04	8.61	1.38	1.11	2.72

Table 1.1. Execution times in seconds

Table 1.1 provides the execution time for the benchmarks. The second and third columns of Table 1.1 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 (OSICS),
- the modified Prolog program run with SICStus Prolog (SICS),
- the Prolog-equivalent HAL program (obtained with the preprocessor) which containts no precise declarations (None),
- with precise type declarations (T),
- with precise type declarations and scheduling information (i.e. replacing `deriving delay` by `deriving herbrand` wherever possible) (TS),
- with precise type declarations, scheduling information, and mode declarations (TSM),
- with precise type declarations, scheduling information, and mode and determinism declarations (TSMD),
- this last version run with Mercury (if possible) compiled with trailing support (Merc+tr),
- the same Mercury version without trailing support (Merc).

[16] See e.g. `http://www.puzzlemuseum.com/griddler/griddler.htm`

The last row of the table contains the geometric mean speed ratio between the preceeding column and the current column. For example, programs in the TSM column are, on average, 8.61 times as fast as the corresponding program in the TS column.

The benchmarks `nono` and `bqueens` use dynamic scheduling code which is required to be `semidet`. Hence, we required some modification of the original code to ensure that the determinism was checkable by the compiler for versions before TSMD.

In general, the original and modified SICStus programs have similar speed. `deriv` slows down because of loss of indexing caused by the introduction of if-then-elses, 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 are mostly slower than the modified SICStus versions. Slow-down 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. For the benchmarks with delay, since the scheduling strategies are impossible to make the same, the comparison is rather meaningless.

Generally, adding precise type information leads to a slow down (on average 0.77 times as fast). For the version with no information, we used a monomorphic "universal" type which included all the functors in the program. For the version with type information, we use the polymorphic types where appropriate. The slow down is due to the use of polymorphic unification predicates. The compiler could remove this cost by providing type specialized versions of these predicates (indeed if we use only non-polymorphic types the relative performance is 1.33 in favour of types). The programs `fib` and `tak` do not use polymorphic types and therefore do not incur this cost. We see improvements for both of these benchmarks. For `warplan` we gain a large improvement because it allows a type specialized version of univ to be used.

Adding precise scheduling information provides a modest improvement for most of the benchmarks (average 1.04 times). It provides no improvement for `bqueens` and `nono`, both of which make extensive use of dynamic scheduling.

Adding mode declarations provides the most speed-up (on average 8.61 times). 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. Interestingly `bqueens` obtains no speedup since the bulk of the time is in the search, using the dynamic scheduling, and this is unchanged. For `nono` the dynamic scheduled code is itself complex, and so benefits from mode information.

Determinism declarations also lead to significant speed-up (on average 1.38 times). Again the benchmarks with dynamic scheduling are the least affected, since the search dominates.

The times given in final three columns of Table 1.1 are too small to make a meaningful comparison. For that reason, Table 1.2 shows the execution times for 100 repeats of each benchmark. We omit `bqueens`, `hanoidiff`, `nono`, `qsortdiff`, `serialize` and `warplan` since their final HAL versions still need herbrand types.

Benchmark	TSMD	Merc+tr	Merc
aiakl	4.85	4.3	3.55
boyer	9.37	10.53	9.97
deriv	79.73	76.02	35.52
fib	2.61	2.61	1.17
hanoiapp	40.07	40.15	34.78
mmatrix	5.27	4.99	4.99
qsortapp	32.79	33.25	24.23
tak	6.06	6.35	4.2
Average		1.01	1.40

Table 1.2. Execution times in seconds for 100 repeats

Benchmark	None	TS	TSM	TSMD	Merc
aiakl	3637	2641	0	0	0
boyer	4904	4904	0	0	0
bqueens	3562	3581	3446	3446	—
deriv	40530	40530	0	0	0
fib	1897	1897	0	0	0
hanoiapp	72704	72704	0	0	0
hanoidiff	7168	7168	6144	6144	—
mmatrix	7970	7970	0	0	0
nono	953	953	307	307	—
qsortapp	51449	51449	0	0	0
qsortdiff	51126	51126	352	352	—
serialize	17244	17244	1552	1552	—
tak	5173	5173	0	0	0
warplan	34	34	2	2	—

Table 1.3. Memory usage in Kbytes for the Trail

The HAL version running with precise declarations is very similar to the Mercury version with trailing support. When we compile the Mercury version without trailing support we see an improvement of 1.4 times on average.

Benchmark	None	TS	TSM	TSMD	Merc
aiakl	2712	38498	1231	1231	1231
boyer	5948	5950	3561	3561	3561
bqueens	81074	641074	101074	101074	—
deriv	27712	27712	24949	24949	24949
fib	2371	2371	0	0	0
hanoiapp	41472	438783	37888	36864	36864
hanoidiff	6656	20480	57344	57344	—
mmatrix	19610	47659	79	79	79
nono	641082	641074	641082	641082	—
qsortapp	25842	269666	25607	25490	25490
qsortdiff	25446	261314	28317	28317	—
serialize	8928	90622	8331	8331	—
tak	5173	5173	0	0	0
warplan	23	22	18	18	—

Table 1.4. Memory usage in Kbytes for the Heap

Benchmark	None	TS	TSM	TSMD
aiakl	<1 (2)	0 (1)	0 (1)	0 (1)
boyer	<1 (2)	<1 (2)	0 (1)	0 (1)
bqueens	80 (154)	85 (154)	100 (154)	100 (154)
deriv	<1 (129)	<1 (129)	0 (1)	0 (1)
fib	0 (1)	0 (1)	0 (1)	0 (1)
hanoiapp	0 (1)	0 (1)	0 (1)	0 (1)
hanoidiff	25 (2)	25 (2)	100 (2)	100 (2)
mmatrix	<1 (2)	0 (1)	0 (1)	0 (1)
qsortapp	0 (1)	0 (1)	0 (1)	0 (1)
qsortdiff	<1 (2)	<1 (2)	100 (2)	100 (2)
serialize	1 (18)	1 (18)	100 (18)	100 (18)
tak	0 (1)	0 (1)	0 (1)	0 (1)
warplan	<1 (4)	1 (4)	99 (4)	99 (4)

Table 1.5. Percentage of chains with more than one element, and maximum chain

We have also investigated the effect of the declarations on memory usage. Table 1.3 shows the trail usage for each benchmark, whereas Table 1.4 shows heap usage. The size of the trail is mostly affected by the presence or absence of precise mode declarations. Adding precise mode declarations greatly reduces trail size — only those benchmarks with Herbrand solver types may need to use the trail.

In many cases, adding precise type definitions causes a significant increase in heap usage. This is due to the use of polymorphic data types. The unification predicates for such types construct data structures for run time type

information on the heap, and the affected benchmarks make many calls to these predicates.

Adding precise modes causes a significant reduction in heap size for most benchmarks. This is mainly because most of the calls to the unification predicates can be removed. It is also no longer necessary to box primitive types, such as `int`s and `float`s. For example, without such boxing `fib` and `tak` use no heap space at all.

Finally, we have investigated the size of the alias cycles constructed using PARMA bindings. The results are shown in Table 1.5. Virtually all cycles have length one immediately before being bound to a non-variable term. Only four benchmarks, `bqueens`, `deriv`, `warplan` and `serialize`, have a maximum cycle length of more than two (154, 129, 4 and 18 respectively). The cycles disappear for `deriv` with mode information. The percentage of non unit cycles dramatically increases for `qsortdiff` and `serialize` with the addition of mode information. The number of non unit cycles does not increase; rather, the number of unit cycles is reduced to zero because the addition of mode information allows us to remove the `deriving herbrand` declarations for some types, meaning that we do not use PARMA chains when binding variables of those types.

1.8 Related Work

As far as we know, HAL is the first logic programming implementation to use the PARMA variable representation and binding scheme since it was introduced in [12]. We note that [8] 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; 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 [9], 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. [13] 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, [12] 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 1.7) with those found for Aquarius and PARMA. One problem is the differences between

the underlying abstract machines and the optimizations performed by each compiler. 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. Another problem 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.

1.9 Conclusions

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. This means that the compiler is free to use the more efficient Mercury term manipulation operations whenever this is possible.

There are however a number of ways to improve HAL's Herbrand constraint solving which we shall investigate. These include better tracking of where one-step dereferencing may be (or rather, is not) required, and 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 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.

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, 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. We would also like to thank David G. Jeffery, Nick Nethercote and Peter Schachte.

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.J. Stuckey. An overview of HAL. In J. Jaffar, editor, *Proceedings of the Fourth International Conference on Principles and Practices of Constraint Programming*, LNCS, pages 174–188. Springer-Verlag, October 1999.
3. T. Dowd, P. Schachte, F. Henderson, and Z. Somogyi. Using impurity to create declarative interfaces in Mercury. Technical Report 2000/17, Department of Computer Science, University of Melbourne, 2000. http://www.cs.mu.oz.au/research/mercury/information/papers.html.
4. M. García de la Banda, P.J. Stuckey, W. Harvey, and K. Marriott. Mode checking in HAL. In J. LLoyd et al., editor, *Proceedings of the First International Conference on Computational Logic*, LNCS 1861, pages 1270–1284. Springer-Verlag, July 2000.
5. Programming Systems Group. *SICStus Prolog User's Manual*, release 3.11.0 edition, 2003.
6. C. Holzbaur. Metastructures vs. attributed variables in the context of extensible unification. In *Proceedings of the International Symposium on Programming Language Implementation and Logic Programming*, number 631 in LNCS, pages 260–268. Springer-Verlag, 1992.
7. D. Jeffery, F. Henderson, and Z. Somogyi. Type classes in mercury. Technical Report Technical Report 98/13, Department of Computer Science, University of Melbourne, Melbourne, Australia, 1998.
8. T. Lindgren, P. Mildner, and J. Bevemyr. On Taylor's scheme for unbound variables. Technical report, UPMAIL, October 1995.
9. 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.
10. K. Marriott and P.J. Stuckey. *Programming with Constraints: an Introduction*. MIT Press, 1998.
11. Z. Somogyi, F. Henderson, and T. Conway. The execution algorithm of Mercury: an efficient purely declarative logic programming language. *Journal of Logic Programming*, 29:17–64, 1996.
12. A. Taylor. PARMA–bridging the performance gap between imperative and logic programming. *Journal of Logic Programming*, 29(1–3), 1996.
13. P. Van Roy. Can Logic Programming Execute as Fast as Imperative Programming? Report 90/600, UCB/CSD, Berkeley, California 94720, Dec 1990.
14. D. H. D. Warren. An abstract Prolog instruction set. Technical Report 309, SRI International, Menlo Park, U.S.A., Oct. 1983.