Compiling the HAL variable to Mercury

Bart Demoen
María García de la Banda
Warwick Harvey
Kim Marriott
Peter Schachte
Peter Stuckey

Report CW 273, September 1998

Katholieke Universiteit Leuven
Department of Computer Science
Celestijnenlaan 200A – B-3001 Heverlee (Belgium)

Compiling the HAL variable to Mercury

Bart Demoen
María García de la Banda
Warwick Harvey
Kim Marriott
Peter Schachte
Peter Stuckey

Report CW 273, September 1998

Department of Computer Science, K.U.Leuven

Abstract

HAL is a new logic language that makes it easy to implement constraint solvers. HAL gets most of its efficiency from compiling to Mercury code. The main mismatch between HAL and Mercury is that HAL supports variables a la Prolog, while Mercury recognises basically only the instantiations new and ground. We describe here the schema that overcomes this mismatch: it relies on a Parma representation of variables. Its main advantage is that once a datastructure is ground, it has the same internal representation as a Mercury ground term. Experiments show that our schema is very competitive with any other logic implementation that supports unbound terms. We also discuss the implementation of delay for the Herbrand solver.

Compiling the HAL variable to Mercury [*]

Bart Demoen, María García de la Banda, Warwick Harvey,
Kim Marriott, Peter Schachte and Peter Stuckey
Department of Computer Science, Katholieke Universiteit Leuven, Belgium
Department of Computer Science, University of Monash, Australia
Department of Computer Science, University of Melbourne, Australia

Abstract

HAL is a new logic language that makes it easy to implement constraint solvers. HAL gets most of its efficiency from compiling to Mercury code. The main mismatch between HAL and Mercury is that HAL supports variables a la Prolog, while Mercury recognises basically only the instantiations new and ground. We describe here the schema that overcomes this mismatch: it relies on a Parma representation of variables. Its main advantage is that once a datastructure is ground, it has the same internal representation as a Mercury ground term. Experiments show that our schema is very competitive with any other logic implementation that supports unbound terms. We also discuss the implementation of delay for the Herbrand solver.

1 Introduction

Mercury relies on mode information - either inferred or provided by the user - to obtain high speed [5]. Modes describe a transition from one instantiation of a term to another instantiation, where an instantiation is a description of the extent to which a term is bound. Mercury distinguishes basically only between *new* and *ground*; new means that the term is not bound and has no other occurences. This last restriction means that in Mercury one can build a list of (distinct) free variables, but one cannot use the elements of this list. This precludes the classical use of difference lists in Mercury. In practice, this restriction is overcome by reordering goals in a clause (performed by Mercury) or by rewriting the program. However, a smoother transition from Prolog to Mercury would be possible if a clean version of the Prolog variable were present in Mercury. Clean meaning: without the non-logical predicates var/1, nonvar/1 ...

We are also interested in Mercury supporting a Prolog variable from the point of view of the HAL project [3]: HAL is a logic language currently under development at Monash and Melbourne which offers support for writing constraint solvers. The implementation of HAL is by way of compiling HAL code to Mercury. HAL supports a richer set of instantiations than Mercury does. It is entirely possible to treat a set of terms - a type - which allows the Prolog variable style of instantiation, as just another solver type, i.e. such terms could belong to a solver over Herbrand terms and be treated by the compilation process as any other solver. The drawback is that this solver needs better support from within the HAL implementation in order to be reasonably efficient, meaning "at least as fast as a fast current Prolog system".

The main requirement for any schema that enhances Mercury with the Prolog variable is that it can co-exist with the current implementation from Melbourne, i.e. we want to reuse most of the

[*]This is a HAL working document - started July 1997, finished September 1998

implementation technology of Mercury team. Moreover, the schema should have minimal impact on the effeciency of Mercury when the logical variable is not used in a particular program.

Our schema employes the PARMA ([6]) representation of unbound variables. The advantages and drawbacks of this representation compared to a WAM representation ([1]) are discussed in [4]. The main point we cancontribute to this discussion is that within the PARMA representation schema, when a term becomes ground, it has exactly the same internal representation as a ground Mercury term constructed in the usual Mercury way. The advantage of this is huge: no dereferencing is ever needed once a term is known to be ground, so the efficiency of Mercury is within reach for ground terms constructed by the Herbrand solver. Within the WAM representation schema, this would not be true, unless program analysis can derive this.

We have implemented partly the PARMA and WAM schema, just enough to make a comparison on a particular benchmark program. We compare also with Mercury and by doing so indirectly also to Aquarius.

Our schema has the following characteristics:

- it reserves a Mercury tag for the unbound variable
- the representation of unbound variables is based on PARMA
- a trail is required

We assume some knowledge of Prolog implementation in general; see for instance [1]. In section 2 we describe the representation of variables as in PARMA. In section 3 we describe the instantiation *old* which in HAL denotes a possible unbound object. Section 4 describes the implementation of the Herbrand solver. Section 5 discusses some benchmark results and section 6 concludes.

2 The PARMA representation of variables

In PARMA ([6]) a new free variable is represented by a self reference (on the heap). When two free variables are bound to each other, they are made to point to each other. In general, if N free variables have been bound to each other, the representation of this binding is a ring of N heap cells. When such a set of N variables is bound to a non free term, every of the N cells is filled in with the term. The main advantage is that dereferencing is a constant time operation, as external pointers (e.g. argument registers or local environment variables) point directly into the ring.

There are also a few disadvantages:

- checking whether two free variables are different is more involved
- when instantiation a variable chain, (conditional) trailing must occur for every of the N cells in the ring; i.e. binding a variable is always linear in the size of the ring
- when creating a structure which contains an already existing variable, the ring of variables grows (and trailing occurs potentially)

For a more thorough discussion see [4].

From the point of view of HAL, the PARMA binding schema is attractive because after a Prolog object has become ground, its representation is such that dereferencing is not needed: as such, it means that the representation is exactly as if Mercury created the term. Since the HAL compiler transforms HAL code to Mercury code, this is interesting because it allows the writing of programs that work with partially instantiated data for some time (since HAL allows this) and then once the data is ground, process it further with regular Mercury code.

3 A new instantiation: old

4 The compilation of Herbrand unification

Herbrand unification can be specialised when something is known about the mode. Mercury does this and has few cases to consider, since basically only ground and new exist as instantiations. We also consider only two "sorce forms" of unification:

```
        X = Y
and
        X = f(A1,...,An)
```

In the latter one, the allowed instantiation of the Ai depends on the instantiation of X (and possibly on whether f/n belongs to a Herbrand type or not).

We list in the subsequent subsections all cases (apply symmetry if needed).

Note that the correctness of the translation depends on Mercury not reordering goals anymore.

We start with describing the polymorphic lower level predicates. The code is very explicit about the internal representation of Parma variables and is not taking into account secondary tags.

4.1 var/1

Var/1 is used internally in the unification routines: it is at this stage not clear whether var/1 will belong to HAL.

```
:- pred var(T).
:- mode var(oo) is semidet.
:- pragma c_code(var(Var::oo),
        will_not_call_mercury,
"
        { Word Dereffed;
          if ((Var & 0x3) == 0x3)
             { Dereffed = *(Word *)(Var & ~0x3);
               SUCCESS_INDICATOR = ( (Dereffed & 0x3) == 0x3 );
             }
          else SUCCESS_INDICATOR = 0;
        }
").
```

4.2 promise_ground/1

```
:- pred promise_ground(T).
:- mode promise_ground(og) is det.
:- pragma c_code(promise_ground(X::og),
        will_not_call_mercury,
"
").
```

4.3 deref/2

```
:- pred deref(T,T).
:- mode deref(oo,ng) is det.
:- pragma c_code(deref(X::oo,Y::ng),
        will_not_call_mercury,
"
       if ((X & 0x3) == 0x3)
            Y = *(Word *)(X & ~0x3);
       else Y = X;
").
```

4.4 unify_var_var/2

```
:- pred unify_var_var(T,T).
:- mode unify_var_var(oo,oo) is det.
:- pragma c_code(unify_var_var(X::oo,Y::oo),
        will_not_call_mercury,
"
        { Word *PX, QX, QY, *PY, BX, BY;
          BX = X; BY = Y;
          PX = (Word *)(BX & ~0x3);
          PY = (Word *)(BY & ~0x3);
          QX = *PX; QY = *PY;
          /* first determine equality of X and Y */
          /* do this in a symmetrical way */

          check_equal:
          if (QX == BY) goto they_are_equal;
          if (QY == BX) goto they_are_equal;
          if ((QX != BX) && (QY != BY))
               { PX = (Word *)(QX & ~0x3);
                 QX = *PX;
                 PY = (Word *)(QY & ~0x3);
                 QY = *PY;
                 goto check_equal;
               }
          /* they are not equal */

          PX = (Word *)(BX & ~0x3);
          PY = (Word *)(BY & ~0x3);
          MR_trail_current_value(PX);
          MR_trail_current_value(PY);
          QX = *PX;
          *PX = *PY;
          *PY = QX;

          they_are_equal: /* nothing needs to be done */
        }
").
```

4.5 unify_var_val

```
:- pred unify_var_val(T,T).
:- mode unify_var_val(oo,oo) is det.
:- pragma c_code(unify_var_val(Var::oo,Val::oo),
        will_not_call_mercury,
"
        { Word *P, Q;
          P = (Word *)(Var & ~0x3);
          Q = *P;
          while ((Q & 0x3) == 0x3)
             { MR_trail_current_value(P);
               *P = Val;
               P = (Word *)(Q & ~0x3);
               Q = *P;
             }
        }
").
```

4.6 X = Y and X and Y are new

This situation is dealt with in a later document ([2]).

4.7 X = Y and X is new

This is an "assignment" and translated to Mercury as "X = Y" because Mercury generates the right code for HAL.

4.8 X = Y and both are ground

Translate to itself in Mercury.

4.9 X = Y and X is old

Y is not new otherwise a case above applies.

X and Y belong to the same type t which is a Herbrand type. T could be polymorphic. Assume that type t is defined in module m.

Compile to the goal "m:hbu_t_n(X,Y)". hbu_t_n is the unification predicate for the type t (with arity n) and defined as:

```
:- pred hbu_T_n(t,t).
:- mode hbu_T_n(oo,oo) is semidet.

hbu_T_n(X,Y) :-
        (var(X) ->
            (var(Y) ->
                unify_var_var(X,Y)
            ;   unify_var_val(X,Y)
            )
        ;   (var(Y) ->
                unify_var_val(Y,X)
            ;   unify_val_val_t(X,Y)
            )
        ).
```

5 Performance measurements

5.1 nrev

```
:- pred nrev(list(bla),list(bla)).
:- mode nrev(list_oo,(new -> list_of_old)) is det.

nrev(In,Out) :-
        (   In = [],
            Out = []
        ;   In = [X|S],
            nrev(S,R),
            Int = [X],
            get_address(Int,Address),
            copy_address(Address,X),
            my_append(R,Int,Out)

:- pred my_append(list(bla),list(bla),list(bla)).
:- mode my_append(list_oo,list_oo,(new -> list_of_old)) is det.

my_append(In,Tail,Res) :-
        (   In = [],
            Tail = Res
        ;   In = [X|R],
            my_append(R,Tail,NewR),
            Res = [X|NewR],
            get_address(Res,Address),
            copy_address(Address,X)
        ).
```

5.2 The figures

All the figures are in MLIPS (mega-lips).

- HAL-g: the list contained non-variable objects
- HAL-f: the list contained (distinct) variable objects

length list	Mercury	HAL-g	HAL-f	SICStus-e	SICStus-n
10	3.35	2.35	2.0	0.46	2.6
100	3.14	1.85	1.37	0.71	3.7
1000	2.58	1.62	1.21	0.74	3.2
30	3.75	2.36	1.94	0.61	2.6

The mode of the HAL nrev and append were declared as $old- > old$ and $new- > old$

- SICStus-e: SICStus in compactcode mode
- SICStus-n: SICStus in fastcode mode (extrapolated)

A strange thing can be noticed from the figures: in WAM (SICStus), the longer the list, the higher the LIPS. This seems not to hold for the MAM (Mercury and HAL). The reason is the lack of real tail recursion optimisation in Mercury: it does a good approximation (middle recursion optimisation) but for long lists, there is a negative effect on speed.

Mercury was also with trail - but this doesn't affect the timings really.

All figures to be taken cum grano salis.

6 Conclusion

Efficiency was not the prime reason for our choice of the PARMA binding schema, but the nrev benchmark indicates that the choice of the PARMA representation for variables within the Mercury schema, is reasonable. Future work will concentrate on implementing fully the described schema, extending it for polymorphic types and optimising it.

Acknowledgements

The HAL team is grateful to the Mercury team for patient discussions. We want to thank in particular, Zoltan Somogyi, Fergus Henderson and Thomas Conway. Most of this work was done while the first author was a guest at the University of Monash and the University of Melbourne. Support from project G.0218.95 of the Belgian FWO is also acknowledged. Also support from the HAL project ARC Large Grant is acknowledged.

References

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

[2] Bart Demoen, María García de la Banda,Warwick Harvey, Kim Marriott, Peter Stuckey *Herbrand Constraint Solving in HAL.* To Appear in the Proceedings of ICLP'99, Las Cruces, New Mexico

[3] Bart Demoen, María García de la Banda, Kim Marriott, Peter Schachte, Peter Stuckey *HAL: a highly attractive language for writing (constraint) solvers* Documentation Manual release 0.1 - June 1997

[4] Thomas Lindgren, Per Mildner, Johan Bevemyr *On Taylor's Scheme for Unbound Variables* UPMAIL Technical Report No. 116, October 25, 1995, ISSN 1100-0686

[5] Zoltan Somogyi, Fergus Henderson and Thomas Conway. *The execution algorithm of Mercury: an efficient purely declarative logic programming language.* Journal of Logic Programming, volume 29, number 1-3, October-December 1996, pages 17-64.

[6] Andrew Taylor *PARMA–Bridging the Performance Gap between Imperative and Logic Programming* Journal of Logic Programming, volume 29, number 1-3, October-December 1996