

Type class support for HAL in hProlog.

Bart Demoen
María García de la Banda
Peter J. Stuckey

Report CW315, June 2001



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

Type class support for HAL in hProlog.

Bart Demoen
María García de la Banda
Peter J. Stuckey

Report CW 315, June 2001

Department of Computer Science, K.U.Leuven

Abstract

As a backend to HAL, hProlog must provide support for type classes. Standard Prolog has enough builtins to make such support possible, but we show how much better performance can be obtained by providing a few new builtin predicates. We also show how the HAL compiler (to hProlog) can generate code such that almost no performance is lost without specializing predicates that are type class constrained. Mercury has solved most of these problems in a very satisfactory way, but the problem within an untyped WAM setting is new. We also discuss the problems of working with type classes in an interactive setting, where debugging and program changes must be possible.

Type class support for HAL in hProlog

Bart Demoen*, Maria García de la Banda†, and Peter Stuckey‡

* Department of Computer Science, K.U.Leuven, Belgium
bmd@cs.kuleuven.ac.be

† School of Computer Science & Software Engineering, Monash University, Australia
mbanda@csse.monash.edu.au

‡ Department of Computer Science and Software Engineering, University of
Melbourne, Australia
pjs@cs.mu.oz.au

Abstract. The HAL language has been recently extended to support type classes. As a result, compilation to its Prolog back-end has also been modified to support type classes. Prolog has enough builtins to make such support readily possible, but the performance penalty seemed high enough to investigate alternatives. The aim of this paper is to experiment with different levels of support in order to reduce the overhead. In particular, we consider transformations at the source-to-source level, at the abstract machine code level, transformations allowed by the introduction of special purpose built-in predicates, and even those allowed by using knowledge about the determinacy of HAL predicates. All experiments were performed using the hProlog language, whose compiler can be modified by the authors. While Mercury has dealt with type classes in a successful way, studying these issues within the Prolog and WAM setting is new.

1 Introduction

Type classes [9, 11] support *constrained* polymorphism by allowing the programmer to write code which relies on a parametric type having certain associated predicates and functions. More precisely, a *type class* is a name for a set of types for which certain predicates, called the *methods*, are defined. Type classes were first introduced in the functional programming languages Haskell and Clean. While Mercury was the first logic programming language to include them [8], HAL has been recently extended to provide type classes similar to those in Mercury. One major motivation for this extension is that type classes provide a natural way of specifying a constraint solver’s capabilities and, therefore, support for “plug and play” with solvers, which is one of the main aims of HAL.

Extending HAL to provide type classes meant that compilation to SICStus Prolog (which has been one of the traditional back-ends of HAL) would have to be modified to support type classes. Prolog has enough builtins to make such support readily possible, but the performance penalty seemed high enough to investigate alternatives. The aim of this paper is to experiment with different

levels of support in order to reduce the associated overhead. In particular, we will consider transformations at the source-to-source level, at the abstract machine code level, transformations allowed by the introduction of special purpose built-in predicates, and even those allowed by using knowledge about the determinacy of HAL predicates. In order to do this we have chosen a new back-end for HAL: the hProlog system. hProlog is a Prolog implementation derived from [4] with support for global variables and delay primitives, but lacks some ISO Prolog features, especially those that are not needed for HAL. The main reason to use hProlog instead of a better known Prolog system is that we have full control over its implementation. While Mercury has dealt with type classes in a successful way, studying these issues within the Prolog and WAM setting is new. One might wonder why Mercury is not enough as a backend for HAL: Mercury lacks the quick turn around while explorative programming. The hope is that with hProlog as a backend, this much appreciated feature becomes available to the HAL programmer.

One approach to dealing with type classes is specialization of code, perhaps by a JIT partial evaluator. The specialization approach has been followed by Mercury for the *int* type, because the performance gains are huge in that case. We will not pursue this path here. The functional programming world has also produced important results in this area that might be relevant for us in the longer run.

The rest of the paper is organized as follows. Section 2 provides a brief introduction to type classes and their naive transformation. Section 3, measures the overhead of each of the components in the naive transformation. In section 4 we try to reduce the overhead that comes from meta-calling the type class predicates. Section 5 we introduces a special purpose variation on meta-call. In section 6 we try to lower the overhead of the selection of the appropriate type class method from the dictionary. In section 7 we exploit semi-det-ness of type class methods. Finally, section 9 concludes.

2 Type classes and their naive transformation

A `class` declaration defines a new type class. It gives the names of the type variables which are parameters to the type class, and the methods which form its interface. For example, one of the most important built-in type classes in HAL is that defining types which support equality testing:

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

Instances of this class can be specified, for example, by the declaration

```
:- instance eq(int) where [ pred(==/2) is intequal ].
```

which declares the `int` type to be an instance of the `eq/1` type class, with the class method `==/2` being implemented by the predicate `intequal/2`.

Type classes can require their arguments to belong to some other class. Consider, for example, the class `ord/1` defined as:

```
:- class ord(T) <= eq(T) where
    [ pred T < T,
      mode in < in is semidet,
      pred T > T,
      mode in > in is semidet].
```

which requires all its instances to also be instances of the class `eq/1`.

Note also that type classes can be multi-parameter, i.e., they can have more than one argument.

Class constraints can appear as part of a predicate's type signature. They constrain the variables in the type signature to belong to particular type classes. Consider, for example, the following program:

```
:- pred qsort(list(T),list(T),list(T)) <= ord(T).
:- mode qsort(in, in, out) is det.
:- pred split(list(T),T,list(T),list(T)) <= ord(T).
:- mode split(in, in,out, out) is det.

qsort([],L,L).
qsort([X|R],Acc,Out) :-
    split(R,X,Small,Large),
    qsort(Large,Acc,LS),
    qsort(Small,[X|LS],Out).

split([],_,[],[]).
split([A|R],Split,Small,Large) :-
    ( A < Split →
      Small = [A|S], split(R,Split,S,Large)
    ; A > Split →
      Large = [A|L], split(R,Split,Small,L)
    ;
      split(R,Split,Small,Large)
    ).
```

which defines two predicates, `qsort/3` (written with an accumulating parameter) and `split/4`. Both predicates are defined in terms of the type parameter `T` which is required to belong to class `ord/1` and thus, by inheritance, to class `eq/1`.

The basic translation schema for predicates whose type declaration contains type class constraints, and for calls to such predicates, is described in [8]. Our running example and (only) benchmark will be the `qsort` program introduced above ¹. This program will be tested on lists of integers. Thus, we will need the following instance declarations:

```
:- instance eq(int) where [ pred (==/2) is intequal ].
:- instance ord(int) where [ pred (</2) is intsmaller, pred (>/2) is intlarger ].
```

We have chosen the names `intequal`, `intsmaller` and `intlarger` for avoiding confusion, but in Prolog, these are implemented by the usual `<`, `>` and `=` operators, of course.

¹ `qsort` is like most sorting routines amenable to restricted polymorphism and it has the advantage of being universally well understood

Now, let us assume that the query or goal `qsort([3,1,4,2,6,5],[],Out)` appears somewhere in the program. The HAL compiler would implement the basic translation defined in [8], by generating the following hProlog code for the query:

```
Dict = ord(intsmaller, intlarger, eq(intequal)),
qsort([3,1,4,2,6,5],[],Out,Dict)
```

and translating the definition of predicates `qsort/3` and `split/4` as follows:

```
qsort([],L,L,-).
qsort([X|R],Acc,Out,Dict) :-
    split(R,X,Small,Larg,Dict),
    qsort(Larg,Acc,LS,Dict),
    qsort(Small,[X|LS],Out,Dict).

split([],-,[],[],-).
split([A|R],Split,Small,Larg,Dict) :-
    Dict = ord(SmallPred, LargePred, _),
    ( call(SmallPred,A,Split) →
      Small = [A|S],
      split(R,Split,S,Larg,Dict)
    ; call(LargePred,A,Split) →
      Large = [A|L],
      split(R,Split,Small,L,Dict)
    ;
      split(R,Split,Small,Larg,Dict)
    ).
```

where the third argument of term `Dict` is the dictionary for the `eq/1` class. Intuitively, the translation means that (a) each type class `name/n` with k methods and l class constraints, has an associated dictionary of the form `name(InfoForMethod1, ..., InfoForMethodk, DictClass1, ..., DictClassl)`, where the particular `info` in `InfoForMethodi` will depend on the type of the instance (b) all predicates with associated class constraints are extended by adding one argument per (non-entailed) class constraint appearing in their type declaration, and (c) each call to a class method is replaced by first a lookup in the dictionary of the associate predicate name, and then a call to this predicate.

Note that we make use of the non-standard — but provided by some implementations — `call/n` predicates. Also note that we have taken apart the dictionary only once in the clause. This optimization can easily be done by the HAL compiler.

We refer to the version obtained by the above translation as the *naive type class version*.

3 Cost of the naive translation schema

We can already make an analysis of the performance loss of the naively translated program compared with a specialized version of `qsort` for integers—this specialized version will look like the HAL code. The overhead of the type class version in Prolog comes from the following sources;

1. an extra argument (the dictionary) needs to be passed around

2. at every entrance of the second clause of `split/4`, the dictionary needs to be inspected
3. a meta-call needs to be performed (since the definition of `call/n` contains a meta-call)
4. the efficient if-then-else (at least in hProlog) is replaced by the creation of a choice-point and on success of the condition, a local cut is executed; if the condition fails, there is full backtracking (not shallow) to the else branch
5. since now the body of `split` contains a general goal, the recursive clause needs an environment, so an `allocate` is executed and the tail call performs a `deallocate`

We can easily write a series of variations of the original `qsort` so that the performance loss caused by each of the above sources can be assessed almost independently. In order to do this we run `qsort` on all permutations of a list of 9 different integers in 3 systems: hProlog, SICStus Prolog 3.8.5 and Yap4.3.0. All measurements were done on a Pentium III, 500MHz, 128 Mb with RedHat Linux. Comparison with SICStus Prolog is natural because SICStus Prolog was a back-end for HAL since the beginning of the project. Comparison with Yap is motivated by the fact that Yap is reputedly fast and it shows that hProlog is not a slow system.

	hProlog	SICStus	Yap
emptyloop (permutations)	150	240	120
original <code>qsort</code>	4890	7470	8470
original + extra argument	5250	7830	9240
original + extra environment	7650	10530	9260
original + env+arg+uni	8820	12270	11410
original + extra choice-point	8280	13620	9540
original + all overhead	10290	18770	12740
naive typeclass	13850	129470(30500)	30450

The different rows correspond roughly to the overhead indicated before:

- **extra argument**: an extra argument is passed around in `qsort` and `split`. Note that this extra argument is always added *after* the original arguments. This is important for avoiding (a) costly argument shuffling and (b) loss of indexing.
- **extra environment**: in the recursive clause of `split/4` we have put as first goal a call to predicate `env/0`, which has just one fact as definition. This causes the clause to have an environment and to save all arguments to the clause in the environment.
- **env+arg+uni**: the idea was actually to only pass an argument and do the unification that would happen if it was used as a dictionary. However, the unification `Dict = ord(SmallPred, LargPred, _)` is only performed completely if the variables `SmallPred` and `LargPred` are used afterwards. Thus, we also needed to put a goal `use(SmallPred, LargPred)` somewhere, which meant that the clause got an environment as well.

- **extra choice-point**: we replaced the goal `A < Split` by the goal `smaller(A, Split)`, and added the definition `smaller(X,Y) :- X < Y`. This causes if-then-else to create a choice-point—but it also causes an environment for `split/4`.
- **all overhead**: means all of the above.
- **naive typeclass**: see the code of this version above; it includes the cost of the meta-call.

The above explanation of the effect on the executed code for each of the additions is valid for hProlog. For SICStus and Yap, we ran the same programs of course, but the effect might be different. For example, it looks from the figures as if Yap already does set the choice-point for the if-then-else in the original `split`—and perhaps an environment as well.

The figures for SICStus and Yap are obtained with the following implementation of `call/3`:

```
call(Name,X,Y) :- Goal =.. [Name,X,Y], call(Goal).
```

This is not the usual definition of `call/3` (we would usually have to concatenate `X` and `Y` to the arguments of the goal passed as first argument), but in this case it is correct because we know that the first argument to `call/3` is always an atom. The figure between brackets for SICStus is obtained by replacing the `call(Goal)` by the goal `prolog:call_module(Goal,user,[])` which bypasses most of the overhead due to modules and goal expansion: this fits in the model of using e.g. SICStus Prolog as a back-end for HAL, because the compiled code is appropriately renamed and in module `user`.

Given the performance results obtained above, it is clear that we must first concentrate on lowering the cost of `call/3` since it is in all systems the largest single overhead. Even in hProlog, where it is lowest of all, it represents 35% of the total cost. Lowering the cost of `call/n` will be the subject of next section. Lowering the other overheads will be tackled later.

4 Getting rid of the overhead of `call/3`

4.1 Multifile predicates

The call to predicate `call/n` is needed to access the instance predicates associated with each the class methods. However, at the moment of using `call/n` we already know the name of the instance predicate. Thus, we could also access this instance predicate by constructing another predicate (one for each class method) indexed by such names, which performs the indirection. In other words, a predicate for each type class method and with one clause per instance in which the first argument would be the associated instance predicate, and the rest would be the required arguments. Lets illustrate this with an example.

Consider the instance declaration for `ord(int)` introduced in previous sections. Then, the HAL compiler would output the following code for it:

```

:- multifile 'typeclass_ord_1<_2'/3.
'typeclass_ord_1<_2'(intsmaller,A,B) :- intsmaller(A,B).

:- multifile 'typeclass_ord_1>_2'/3.
'typeclass_ord_1>_2'(intlarger,A,B) :- intlarger(A,B).

```

where the multifile declaration is used to be able to declare clauses of a predicate across different files. Each instance of class ord/1 would result in one extra clause for such predicates.

The recursive clause of split/4 would now be generated as:

```

split([A|R],Split,Small,Large,Dict) :-
    Dict = ord(SmallPred,LargePred,_),
    ( 'typeclass_ord_1<_2'(SmallPred,A,Split) →
      Small = [A|S], split(R,Split,S,Large,Dict)
    ; 'typeclass_ord_1>_2'(LargePred,A,Split) →
      Large = [A|L], split(R,Split,Small,L,Dict)
    ;
      split(R,Split,Small,Large,Dict)
    ).

```

Together with the naive method of building, passing and selecting from the type dictionary, this results in the following timings (we have repeated some of the old rows, just for ease of comparing):

	hProlog	SICStus	Yap
original qsort	4890	7470	8470
original + all overhead	10290	18770	12740
naive typeclass	13850	30500	30450
specialized type class pred	10860	18220	13510

For the timings, we have added some more definitions for the predicates 'typeclass_ord_1*_*/3 because, in general, there might be more than one instance of the ord type class.

The figures indicate that even without special support, most of the overhead of the type class call itself is removed. Furthermore, this is done in such a way that the effect is portable across systems. This is a very satisfactory result since the generation of the above code is also not too difficult. Let us now show how the idea of multifile predicates can be extended to polymorphic instances.

4.2 Dealing with polymorphic instances.

As mentioned in Section 2, all dictionaries associated to classes eq/1 and ord/1 must be of the form eq(Info_for_=) and ord(Info_for_<,Info_for_>,DictForEq), respectively, where the information for each method will depend on the instance itself. In our qsort program this meant that the ord(int) and eq(int) instances lead to the dictionary ord(intsmaller,intlarger,eq(intequal)) for ord(int).

Let us now consider polymorphic instances, i.e., instances whose arguments are polymorphic types. Consider, for example, the following instance declarations for type `list(T)`:

```
:- instance ord(list(T)) <= ord(T) where [
    pred(</2) is listsmaller,
    pred(>/2) is listlarger].

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

They result in a dictionary for `ord(list(T))` of the form:

```
ord(listsmaller(orddictforT),listlarger(orddictforT),
    eq(listequal(eqdictforT)))
```

where the predicates `listsmaller`, `listlarger`, and `listequal` are explicitly provided with the dictionary arguments they require as additional first arguments. In particular, (and in the context of the above instance) `listsmaller` and `listlarger` are assumed to have a predicate type declaration of the form `(list(T),list(T)) <= ord(T)` and thus they need the `ord/1` dictionary for type parameter T , while `listequal` is assumed to have a predicate type declaration of the form `(list(T),list(T)) <= eq(T)` and thus needs the `eq/1` dictionary for type parameter T . Note that at run-time all type parameters will be instantiated to a ground type and, thus, the associated dictionaries will also be ground.

The above instance declarations also result in the following extra clauses for predicate `'type_class_ord_1_<_2'/3`:

```
'typeclass_ord_1_<_2'(intsmaller,A,B) :- intsmaller(A,B).
'typeclass_ord_1_<_2'(listsmaller(Dict),A,B) :- listsmaller(A,B,Dict).
```

In this way we can make use of first argument indexing. In usual restrictions for typeclasses, there can be at most one instance defined for the same type to depth 1. If the same predicate name appears in more than one instance we can explicitly make them different by an extra level of procedure call.

Let us illustrate the effect of the above transformation on predicate `foo/2`:

```
:- pred foo(T,T) <= ord(T).
foo(L1,L2) :- L1 < L2.
```

and goal `foo(L1,L2)` where each L_i is known to be a list of integers. Since the predicate type declaration of `foo/2` has the class constraint `ord(T)`, we need to pass to `foo/2` an additional argument which is the dictionary for `ord(list(int))`. This dictionary is constructed as follows:

```
EqDictInt = eq(intequal),
OrdDictInt = ord(intsmaller,intlarger,EqDictInt),
EqDictListInt = eq(listequal(EqDictInt)),
```

```

OrdDictListInt = ord(listsmaller(OrdDictInt),
                        listlarger(OrdDictInt), EqDictListInt),
foo(L1,L2,OrdDictListInt)

```

The definition of foo/2 is thus translated to:

```

foo(L1,L2,OrdDict) :-
  OrdDict = ord(Smaller,_,_),
  'typeclass_ord_1<_2'(Smaller,L1,L2).

```

Similarly, the original definition of listsmaller/2 (below) will be translated as follows:

```

:- pred listsmaller(list(T),list(T)) <= ord(T).
listsmaller([],[_|_]).          listsmaller([],[_|_],_).
listsmaller([X|R],[Y|S]) :-    listsmaller([X|R],[Y|S],OrdDict) :-
  X < Y.                        OrdDict = ord(Smaller,_,_),
                                'typeclass_ord_1<_2'(Smaller,X,Y).
listsmaller([X|R],[Y|S]) :-    listsmaller([X|R],[Y|S],OrdDict) :-
  X = Y,                        OrdDict = ord(_,_,EqDict),
  listsmaller(R,S).            EqDict = eq(Equal),
                                'typeclass_eq_1=_2'(Equal,X,Y),
                                listsmaller(R,S,Dict).

```

5 A new call/n and how to use it for type classes.

As mentioned before, the modifications to the naive transformation performed in the previous section are quite satisfactory. There are however a few drawbacks as well, mostly related to how well this approach scales. First, we rely on having perfect indexing; if there are many instances of the same type class, systems might either give us less than perfect indexing or take more time. And second, since systems provide typically indexing on the first argument only, the second up to last arguments need shuffling: this will be worse if the arity is higher. This section tries to remedy this by focusing on low level support.

In WAM, a structured term is represented by a STRUCT tagged pointer to a cell (on the heap) which contains a description of the name/arity of the principal functor of this term. This description can be used to find the code address of the predicate associated to this functor. In the case of hProlog, it is simply an index in a function symbol table, but it could be a pointer or anything else encoding a similar thing. So, it is straightforward to consider this as an integer and tag it appropriately.² To support such a conversion at the hProlog level source language we have introduced a builtin predicate `getf/2`: as first argument, it expects an instantiated structured term; it unifies its second argument with an integer that we name *the internal representation* of the principal functor. It will give constant (and low overhead) access to the code (if present) of the predicate with same name/arity of this term.

² Systems with a too low limit on MAXINT might be slightly disadvantaged here

This internal representation can be used in a new builtin `calli/3` which is most easily explained by an example. Consider the Prolog predicate `name/2` which receives as input an atom and returns the list of associated characters. Then query `?- call(name,asd,L)` can be replaced by the query `?- getf(name(_,_),I), calli(asd,L,I)` which yields the answers: $I = 208, L = [97, 115, 100]$.

The use of an internal representation of a functor for meta-calling results in a speed up that is small in hProlog, but should be considerably larger in other systems because of their slower meta-call. The `calli/3` predicate also puts the arguments already in the right place—in contrast with `call/3` (and the type class predicate method of the previous section) which requires also some argument shuffling.

In hProlog, the internal representation of a functor is independent of whether the functor is a predicate or not and code movement (due to code garbage collection) does not affect it, so the internal representation is a safe alternative for the name/arity of a predicate. This means that we can use it in a type class dictionary. Indeed, we can translate the call to the polymorphic `qsort` and the predicate `split/4` to:

```

getf(_ < _,Smaller),      split3([A|R],Split,Small,Large,Dict) :-
getf(_ > _,Larger),      Dict = ord(SmallPred,LargePred,_),
Dict = ord(Smaller,Larger,eq(_)), ( calli(A,Split,SmallPred) →
qsort(NN,[],_,Dict)      Small = [A|S],
                          split3(R,Split,S,Large,Dict)
                          ...

```

We have implemented this `getf/3` and `calli/3` in hProlog. The timings are as follows:

original	original +	naive	specialized type	internal
original	all overhead	typeclass	class pred	representation
4900	10150	13850	10860	10540

The extra gain over the specialized type class predicates obtained by using the internal representation is small. The implementation effort is also slightly larger: the HAL compiler cannot generate directly the internal representation of course, and one would like not to make the `getf` calls at runtime, so it means that a special construct will have to be recognized by the loader.³ As a result, we have not yet decided to use this method in the future.

However, the experiment was useful, as it indicates somehow the limits of what can be improved at the calling point of the type class predicate. This means that from now on, we can concentrate on the overhead of the other issues: extra environment and choice-point, the passing of the dictionary and the selection of the predicates from the dictionary.

³ hProlog has a compiler that emits code on a file; so a separate loading phase is needed

6 Selection from the dictionary: alternatives to unification

6.1 `arg/3`

The unification `Dict = ord(SmallPred,LargPred,_)` is an overkill: we know that `Dict` is ground and that the unification will not fail; in particular, the principal functor of `Dict` is `ord/3`. We can therefore replace that unification by the goal: `arg(1,Dict,SmallPred)`, `arg(2,Dict,LargPred)` with the additional advantage that these goals can be placed just before `SmallPred` or `LargPred` are needed, as in the code:

```
split([A|R],Split,Small,Large,Dict) :-
    ( arg(1,Dict,SmallPred), 'typeclass_ord_1<_2'(SmallPred,A,Split) →
      Small = [A|S], split(R,Split,S,Large,Dict)
    ;
      arg(2,Dict,LargePred), 'typeclass_ord_1>_2'(LargePred,A,Split) →
      Large = [A|L], split(R,Split,Small,L,Dict)
    ;
      split(R,Split,Small,Large,Dict)
    ).
```

The timings obtained with the above transformation are the following:

	hProlog	SICStus	Yap
original qsort	4890	7470	8470
specialized type class pred	10860	18220	13510
<code>arg/3</code> for selection	11390	18240	14160

In none of the implementations, `arg/3` yields any benefit. We should have known: the number of emulator cycles remains the same, and `arg/3` does some tag testing (on the integer, the functor and the variable) and a range test, all of which we know are redundant. So, it might pay off to implement a specialized `arg/3`, which is translated to one (new) emulator instruction `spec_arg`, with the following arguments:

- an integer indicating which argument to fetch
- where to fetch it from
- in which argument register to put it

Obviously, we have not implemented that instruction in SICStus Prolog and Yap. But for hProlog, the resulting timing is 10300.⁴ This represents a 5% improvement over the best timing up to now, which seems only mildly worthwhile. However, this result can be still improved on if the compiler generates the instruction appropriately: at the moment there is still one emulator cycle too many. If we combine the internal representation with the specialized `arg/3` the resulting timing is 10080, which brings us to the timing obtained for *all overhead*.

⁴ We have hand adapted the abstract machine code, not adapted the compiler—for the experiment, this was enough; until a decision has been taken, no such drastic changes as adapting the compiler are done of course

6.2 Just passing what we need: with a little help from analysis

The dictionary that is passed contains potentially many more methods than those actually used inside the polymorphic predicate. In the case of `qsort`, only the `<` and the `>` of the `ord` class are needed. Thus, instead of passing the whole dictionary, just passing these would be enough and would save us the overhead of selecting from the dictionary. To do so requires a little analysis from the compiler, which is cheap. An alternative way of looking at this is that the selection is hoisted out of the loops formed by the recursive predicate or some SCC of the call graph.

The above approach would result in the predicate `qsort` being defined as:

```
qsort(A,B,C,Dict) :-
    Dict = ord(SmallPred, LargePred, _),
    qsort(A,B,C, SmallPred, LargePred).

qsort([],L,L,-,-).
qsort([X|R],Acc,Out,SmallPred, LargePred) :-
    split(R,X,Small,Larg,SmallPred, LargePred),
    qsort(Larg,Acc,LS,SmallPred, LargePred),
    qsort(Small,[X|LS],Out,SmallPred, LargePred).
```

The timings obtained with the above definition are the following:

	hProlog	SICStus	Yap
dictionary as argument	10860	18220	13510
methods as arguments	10790	17650	13910
with internal representation	9830	—	—

It is clear that the benefit in hProlog and SICStus is small, while in Yap it actually gets a little worse. In general, the benefit depends on the following:

- the number of methods to pass as arguments
- how often each method is needed in a clause
- how *deep* the used methods are in the dictionary
- how deep in the call graph the arguments are passed on

7 Semi-det

When the condition in an if-then-else is a general Prolog goal, systems commonly deal with the construct as follows: `(p(X) -> q(X) ; r(X))` is transformed to `mark(B), (p(X), cut(B), q(X) ; r(X))`

or in terms of XSB WAM:

```
getpbreg 8      % puts the current choice point in reg 8
ortry @alt
set_up_args_for_p
call p/1
```

```

        putpbreg 8      % cut: restores the saved choice point from reg 8
        set_up_args_for_q
        call q/1
        jump @after
@alt: ortrust
        set_up_args_for_r
        call r/1
@after: ...

```

Very often, the condition is known to be semi-det: if it is not, the if-then-else in HAL actually must backtrack over the alternative solutions to the condition (like if/3 in SICStus Prolog). As for the `qsort` example, the type class `ord/1` contains the information that the predicate `</2` is semi-det. Mercury exploits the semi-det property successfully by a method that we have first implemented at the source level: semi-det predicates are treated like deterministic predicates with an extra argument that indicates on return whether the goal succeeded or failed. It can be mimicked as follows:

original code	<code>smaller(X,Y) :- X < Y.</code>	<code>split([A R],Spil, ...) :- (smaller(X,Y) -> ...)</code>
transformed code	<code>smaller(X,Y,Ret) :- (X < Y -> true ; Ret = 0).</code>	<code>split([A R],Spil, ...) :- smaller(X,Y,Ret), (var(Ret) -> ...)</code>

We choose the transformation in such a way that the `var/1` test can be used: this check is cheaper than checking for equality with some known value—at least in hProlog.

If the implementation for an if-then-else with a call to `</2` as condition is compiled without a choice-point (and likewise for `var/1`), the transformed code has a good chance of running faster.

internal rep	10500
internal repr + return success	11390

The experiment shows that we have not improved things and this may not be a promising direction. Still, we thought that a little low-level support would be beneficial and we introduced two new builtin predicates:

- `set_success/0`: succeeds and sets a new abstract machine flag to TRUE
- `test_success/0`: succeeds if the above flag is TRUE and sets it to FALSE; otherwise fails

The flag is initially set to FALSE. We then transform the above code for `smaller/2` and `split/4` as follows:

transformed code	<pre> smaller(X,Y) :- (X < Y -> set_success ; true). </pre>	<pre> split([A R],Spil, ...) :- smaller(X,Y,Ret), (test_success -> ...). </pre>
------------------	--	---

We made the compiler aware that `test_success` is one of those predicates for which if-then-else can be translated without a choice-point (like `</2`, `@</2`, `var/1` ...) and re-run the benchmark in the transformed mode. The result in the next table indicates a good potential.

internal repr	10500
internal repr + return success	11390
internal repr + low level return success	10010

Thus, we started looking for ways of using the semi-det property in combination with the above low-level flag idea. As an example, we worked out a (non-polymorphic) `listsmaller/2` predicate, which works on lists of integers and which is semi-det.

original code	transformed code
<pre> listsmaller([],[_ _]). listsmaller([A As],[B Bs]) :- (A < B -> true ; A = B, listsmaller(As,Bs)). </pre>	<pre> listsmaller([],L2) :- L2 \$= [_ _]. listsmaller([A As],L2) :- L2 \$= [B Bs], (test_success -> (A < B -> set_success ; A \$= B, (test_success -> listsmaller(As,Bs) ; true)) ; true). </pre>

The above transformation takes into account that the arguments are instantiated to closed lists. We have also used a new unification procedure `$=/2` which always succeeds but which sets the FLAG according to the success of the ordinary unification of its two arguments (and which in case of failure also undoes bindings if necessary). If adopted, it can give the same sort of specialized unification instructions as in WAM.

The transformation schema is completed by showing the transformation of a call to `listsmaller/2` in a condition:

original code	transformed code
(listsmaller(L1,L2) ->	listsmaller(L1,L2),
asd	(test_success ->
;	asd
qwe	;
)	qwe
)

This successfully avoids putting a choice-point for the if-then-else in which the condition is semi-det. However, we pay a price in the recursive definition of `listsmaller`: it results in more emulator cycles, so we did not improve the code. This seems to indicate that we have tackled the problem from the wrong angle: we tried to avoid the creation of a choice-point but, in doing so, we put even more overhead elsewhere. Instead, we should try to make the choice-point creation cheap while not incurring any new overhead. This can indeed be done, once we know the condition is semi-det, by introducing two new instructions in the WAM:

- **ortrycall** has as arguments a failure continuation (`alt`) and the argument of a call instruction (`p/1` say); as action: combined **ortry alt** and **call p/1**
- **cut1** has no arguments; cut one choice-point and continue

The code corresponding to the XSB WAM code earlier would be:

```

set_up_args_for_p
ortrycall alt p/1
cut1
set_up_args_for_q
call q/1
jump @after
@alt: ortrust
set_up_args_for_r
call r/1
@after: ...

```

We have not implemented the generation of these instructions in hProlog, but the emulator can execute them, so we adapted some generated code by hand. In particular, for a very artificial tight loop we can show good speed-up and, at the same time, some limit on the improvement that seems possible. The loop calls the goal `(test -> true ; true)` repeatedly:

original compilation schema	5260
with ortrycall	4420

If we know a little more about the semi-det condition, we can do even better. For example, if the condition does not affect `TR`, we can even replace the **ortrust** by a **cut1** and the set up of a complete choice-point can be replaced by just a pushing of a failure continuation on the choice-point stack.

This seems a better path to follow: the cost at the call site of the semi-det predicate is reduced and no overhead in the recursive predicate—which WAM already deals with in an efficient way—is introduced. Note also that this exploitation of semi-det predicates has a wider applicability than in the context of type classes, which is just where it started off for us.

8 One more example: a polymorphic implementation of a method

Assume the *ord/1* class and the following instance:

```
:- instance ord(floop) where [ pred(</2) is @< ].
```

Suppose also that @< has a polymorphic pred declaration:

```
:- pred @<(T,T) <= comparable(T).
```

It is clear that if *X* and *Y* are of type *floop*, the goal $X < Y$ must lead to a call $@ < (X, Y, ComparableDict)$. This will fit in the above schema, by introducing a new predicate *floop_<_2/2* with definition:

```
floop_<_2(X,Y) :- construct_comparable_dict(ComparableDict), @<(X,Y,ComparableDict).
```

and replacing the above instance declaration by

```
:- instance ord(floop) where [ pred(</2) is floop_<_2 ].
```

9 Conclusion

We have described some attempts to generate efficient code for HAL code using type classes. Some low-level support seems beneficial, but this work is clearly not finished: one issue to consider in future work is that the translation schema must allow for efficient interactive manipulation of HAL programs, i.e. redefining predicates, interfaces, type classes etc.

References

1. H. Ait-Kaci. *Warren's Abstract Machine: A Tutorial Reconstruction*. The MIT Press, Cambridge, Massachusetts, 1991. See also: <http://www.isg.sfu.ca/~hak/documents/wam.html>.
2. M. Carlsson. *Design and Implementation of an Or-Parallel Prolog Engine*. PhD thesis, The Royal Institute of Technology (KTH), Stockholm, Sweden, Mar. 1990. See also: <http://www.sics.se/isl/sicstus.html>
3. V. S. Costa. *Optimising Bytecode Emulation for Prolog*. Proceedings of PPDP'99, LNCS 1702, Springer-Verlag, 261-277, September, 1999. See also <http://www.ncc.up.pt/~vsc/Yap/>.

4. B. Demoen, P.-L. Nguyen. *So many WAM variations, so little time.* Computational Logic - CL2000, First International Conference, London, UK, July 2000, Proceedings (V. John Lloyd, ed.), Lecture Notes in Artificial Intelligence, vol. 1861, Springer, 2000, pp. 1240-1254.
5. Bart Demoen, Maria García de la Banda, Warwick Harvey, Kim Marriott, Peter Stuckey. *An Overview of HAL* Proceedings of the International Conference on Principles and Practice of Constraint Programming, Oct. 1999, Virginia, USA, pages 174–188, Springer Verlag
6. D. Diaz and P. Codognet. *GNU Prolog: beyond compiling Prolog to C* Proceedings of the Second International Workshop, PADL 2000, Boston, MA, USA, January 2000. LNCS 1753, pp. 81-92 See also <http://gprolog.inria.fr>
7. hProlog: see <http://www.cs.kuleuven.ac.be/~bmd/hProlog>
8. David Jeffery, Fergus Henderson and Zoltan Somogyi. *Type classes in Mercury.* Technical Report 98/13, Department of Computer Science, University of Melbourne, Melbourne, Australia, September 1998, 22 pages.
9. Stefan Kaes. *Parametric overloading in polymorphic programming languages.* In 2nd European Symp. on Programming, volume 300 of Lecture Notes in Computer Science, pages 131–144. Springer, 1988.
10. Ruben Vandeginste and Bart Demoen. *The implementation of a new segment preserving and/or (multi-)generational copying garbage collection for a WAM and its approximation.* K.U.Leuven CW report 319, July 2001 See <http://www.cs.kuleuven.ac.be/publicaties/rapporten/cw/CW319.abs.html>
11. Philip Wadler and Stephen Blott. *How to make ad-hoc polymorphism less ad hoc.* In Proceedings of the 16th ACM Annual Symposium on Principles of Programming Languages (POPL'89), pages 6076. ACM Press, January 1989.
12. D. H. D. Warren. *An Abstract Prolog Instruction Set.* Technical Report 309, SRI International, Menlo Park, U.S.A., Oct. 1983.