Building Constraint Solvers with HAL

María García de la Banda[1], David Jeffery[1], Kim Marriott[1], Nicholas Nethercote[2], Peter J. Stuckey[2], and Christian Holzbaur[3]

[1] School of Comp. Sci. & Soft. Eng., Monash University, Australia.
{mbanda,dgj,marriott}@csse.monash.edu.au
[2] Dept. of Comp. Sci. & Soft. Eng., University of Melbourne, Australia.
{njn,pjs}@cs.mu.oz.au
[3] Dept. of Medical Cybernetics and Art. Intel., University of Vienna, Austria
christian@ai.univie.ac.at

Abstract. Experience using constraint programming to solve real-life problems has shown that finding an efficient solution to a problem often requires experimentation with different constraint solvers or even building a problem-specific solver. HAL is a new constraint logic programming language expressly designed to facilitate this process. In this paper we examine different ways of building solvers in HAL. We explain how type classes can be used to specify solver interfaces, allowing the constraint programmer to support modelling of a constraint problem independently of a particular solver, leading to easy "plug and play" experimentation. We compare a number of different ways of writing a simple solver in HAL: using dynamic scheduling, constraint handling rules and building on an existing solver. We also examine how external solvers may be interfaced with HAL, and approaches for removing interface overhead.

1 Introduction

There is no single best technique for solving combinatorial optimization and constraint satisfaction problems. Thus, constraint programmers would like to be able to easily experiment with different constraint solvers and to readily develop new problem-specific constraint solvers. The new constraint logic programming (CLP) language HAL [3] has been specifically designed to allow the user to easily experiment with different constraint solvers over the same domain, to support extension of solvers and construction of hybrid solvers, and to call procedures (in particular, solvers) written in other languages with little overhead.

In order to do so, HAL provides semi-optional type, mode and determinism declarations for predicates and functions. These allow the generation of efficient target code, ensure that solvers and other procedures are being used in the correct way, and facilitate efficient integration with foreign language procedures. Type information also means that predicate and function overloading can be resolved at compile-time, allowing a natural syntax for constraints. To facilitate writing simple constraint solvers, extending existing solvers and combining them, HAL provides dynamic scheduling by way of a specialized delay construct which supports definition of "propagators." Finally, HAL provides "global variables" which allow efficient implementation of a persistent constraint store. They behave in a similar manner to C's static variables and are only visible within the module in which they are defined.

The initial design of HAL was described in [3]. The current paper extends this in five main ways. First, we describe how the addition of type classes provides a natural way of specifying a constraint solver's capabilities and, therefore, support for "plug and play" with solvers. Second, we give a more detailed description of how HAL supports solver dependent dynamic scheduling and its use in writing solvers. Third, we describe how to integrate foreign language solvers into HAL and some programming techniques for reducing the runtime overhead of the solver interface. Fourth, we discuss the integration of constraint handling rules (CHRs) into HAL. Finally, we compare the efficiency of solvers written in HAL using CHRs, dynamic scheduling and type classes with comparable solvers written in SICStus and compare the overhead of HAL's external solver interface for CPLEX with that of ECLiPSe.

Thus, the main focus of the current paper is how to provide generic mechanisms such as type classes, dynamic scheduling and CHRs for structuring, writing and extending constraint solvers in a constraint programming language with type, mode and determinism declarations and what, if any, performance advantage is provided by this additional information.

2 The HAL Language

In this section we provide a brief overview of HAL [3], a CLP language which is compiled to the logic programming language Mercury [15].[1] The basic HAL syntax follows the standard CLP syntax, with variables, rules and predicates defined as usual (see, e.g., [14] 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 of the objects that it wishes to be visible to those importing the module. Selective importation is also possible. The core language supports the basic integer, float, string, and character data types plus polymorphic constructor types (such as lists) based on these basic types. This support is, however, limited to assignment, testing for equality, and construction and deconstruction of ground terms. More sophisticated constraint solving is provided by importing a constraint solver for each type involved.

As a simple example, the following program is a HAL version of the now classic CLP program `mortgage`.

```
:- module mortgage.                                            (L1)
:- import simplex.                                             (L2)
:- export pred mortgage(cfloat,cfloat,cfloat,cfloat,cfloat).   (L3)
```

[1] The key difference between them is that Mercury does not support constraints and constraint solvers. In fact, Mercury only provides a limited form of unification.

```
:-        mode mortgage(oo,oo,oo,oo,oo) is nondet.                (L4)
mortgage(P,0.0,I,R,P).                                            (R1)
mortgage(P,T,I,R,B) :- T >= 1.0, mortgage(P+P*I-R,T-1.0,I,R,B).   (R2)
```

Line ($L1$) states that this file defines the module `mortgage`. Line ($L2$) imports a module called `simplex`. This module provides a simplex-based linear arithmetic constraint solver for constrained floats, called `cfloats`. Line ($L3$) exports the predicate `mortgage` which takes five `cfloats` as arguments. This is the *type* declaration for `mortgage`. A type specifies the representation format of a variable. Thus, for example, the type system distinguishes between constrained floats (`cfloat`) and standard numerical floats (`float`) since they have a different representation. Types are defined using (polymorphic) regular tree type statements. For instance, the `list/1` constructor type is defined by

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

Line ($L4$) provides a *mode* declaration for `mortgage`. Mode declarations associate a mode with each argument of a predicate. A mode is a mapping $Inst_1 \rightarrow Inst_2$ where $Inst_1$ and $Inst_2$ describe the instantiation of an argument on call and on success from the predicate, respectively. The *base* instantiations are `new`, `old` and `ground`. Variable X is `new` if it has not been seen by the constraint solver, `old` if it has, and `ground` if X is constrained to take a fixed value. Note that `old` is interpreted as `ground` for variables of non-solver types (i.e., types for which there is no solver). 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`$\rightarrow$`ground`. The standard modes `in` and `out` are renamings of `gg` and `ng`, respectively. Therefore, line ($L4$) declares that each argument of `mortgage` has mode `oo`, i.e., takes an `old` variable and returns an `old` variable.

More sophisticated instantiations (lying between `old` and `ground`) may be used to describe the state of complex terms. Instantiation definitions look like type definitions. For example, the instantiation definition

```
:- instdef fixed_length_list -> ([] ; [old | fixed_length_list]).
```

indicates that the variable is bound to either an empty list or a list with an `old` head and a tail with the same instantiation.

Line ($L4$) also states the determinism for this mode of `mortgage`, i.e., how many answers it may have. We use the Mercury hierarchy: `nondet` means any number of solutions; `multi` at least one solution; `semidet` at most one solution; `det` exactly one solution, `failure` no solution and `erroneous` a run-time error.

The rest of the file contains the standard two rules defining `mortgage`.

3 Constraint Solvers and Type Classes

Type classes [13, 17] 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 and/or functions, called the *methods*, are defined. Type classes were first introduced in functional programming languages Haskell and Clean, while Mercury and CProlog were the first logic programming languages to include them [12, 4]. We have recently extended HAL to provide type classes similar to those in Mercury. One major motivation is that they provide a natural way of specifying a constraint solver's capabilities and, therefore, support for "plug and play" with solvers.

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. As an 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 oo = oo is semidet ].
```

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

```
:- instance eq(int).
```

which declares the `int` type to be an instance of the `eq/1` type class. For this to be correct, the module must either define the method `=/2` with type `int=int` and mode `oo=oo is semidet` in the current module or indicate that it is a renaming of some other predicate.

We note that all types in HAL (like Mercury) have an associated "equality" for modes `in=out` and `out=in`, since these correspond to an assignment. Most types also support testing for equality, the main exception being for types that contain higher-order predicates. Thus, by default, HAL automatically generates instance declarations of the above form and the definition of `=/2` methods for all constructor types which contain types supporting equality.

Type classes 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 true equality. Thus, we define the `solver/1` type class to be:

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

This indicates that the `solver/1` type class provides an initialisation method `init/1` and 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 `init/1` and `=/2` for this type with mode and determinism as shown.

Constructor types can be automatically declared to be instances of the `solver/1` type class using the notation `deriving solver`. The compiler then automatically generates an appropriate instance declaration and the predicate `init/1`. Variables whose type is not an instance of `solver/1` are not true logic variables, i.e., they are like Mercury terms since they must either be `new` or bound to a functor of the type. Thus, the type declaration given earlier for lists defines Mercury terms which have a fixed length while

```
:- typedef hlist(T) -> []; [T|hlist(T)] deriving solver.
```

defines true "Herbrand" lists.

Class constraints can appear as part of a predicate or function's type signature. They constrain the variables in the type signature to belong to particular type classes. Class constraints are checked and inferred during the type checking phase except for those of type classes `solver/1` and `eq/1` which must be treated specially because they might vary for different modes of the same predicate. In the case of `solver/1`, this will be true if the HAL compiler inserts appropriate calls to `init/1` for some modes (those in which the argument is initially `new`) but not in others. In the case of `eq/1`, this will be true if equalities are found to be assignments or deconstructions in some modes but true equalities in others. As a result, it is not until after mode checking that we can determine which variables in the type signature should be instances of `eq/1` and/or `solver/1`. Unfortunately, mode checking requires type checking to have taken place. Hence, the HAL compiler includes an additional phase after mode checking, where newly inferred `solver/1` and `eq/1` class constraints are added to the inferred types of procedures for modes that require them. Note that, unlike for other classes, if the declared type for a predicate does not contain the inferred class constraints, this is not considered an error, unless the predicate is exported.[2]

To illustrate the problem, consider the predicate

```
:- pred append(list(T),list(T),list(T)).
:- mode append(in,in,out) is det.
:- mode append(in,out,in) is semidet.
append([],Y,Y).
append([A|X1], Y, [A|Z1]) :- append(X1,Y,Z1).
```

During mode checking, the predicate `append` is compiled into two different procedures, one for each mode of usage (indicated by the keyword `implemented_by`). Conceptually, the code after mode checking is

```
:- pred append(list(T),list(T),list(T)) implemented_by [append_1, append_2].
:- mode append_1(in,in,out) is det.
append_1(X,Y,Z) :- X =:= [], Z := Y.
append_1(X,Y,Z) :- X =: [A|X1], append_1(X1,Y,Z1), Z := [A|Z1].
:- mode append_2(in,out,in) is semidet.
append_2(X,Y,Z) :- X =:= [], Y := Z.
append_2(X,Y,Z) :- X =: [A|X1], Z =: [B|Z1], A =:= B, append_2(X1,Y,Z1).
```

where `=:=`, `:=`, `=:` indicate calls to `=/2` with mode `(in,in)`, `(out,in)` and `(in,out)`, respectively. It is only now that we see that for the second mode the parametric type `T` must allow equality testing (be an instance of the `eq/1` class), because we need to compare `A` and `B`. Thus, in an additional phase of type inference the HAL compiler infers

```
:- pred append_2(list(T),list(T),list(T)) <= eq(T).
```

[2] Exported predicates need to have all their information available to ensure correct modular compilation. We plan to remove this restriction when the compiler fully supports cross module optimizing compilation [1].

Any predicates calling `append_2` will also inherit the `eq(T)` class constraint in their type.

This is a new problem for type classes and multi-moded predicates which does not arise in functional programming. While the same problem arises in Mercury for equality, it is side-stepped by not supporting an anologue of the `eq/1` class: effectively all types are required to support equality for mode `=(in,in)`. This may lead to run-time errors (e.g., when using `append_2` on lists of predicates). Since such errors are caught at compile-time by our two phase scheme, we believe our approach provides a better solution.

HAL provides a hierarchy of pre-defined type classes for common constraint domains which derive from the `solver` type class: `bool_solver`, `linfloat_solver`, `float_solver`, `linint_solver`, and `int_solver`. They provide a standard interface to solvers, thus facilitating "plug and play" experimentation by allowing separate compilation of the constraint models from the solvers that they use. As a result, we can rewrite the type declaration for `mortgage` to

```
:- export pred mortgage(T, T, T, T, T) <= float_solver(T).
```

thus allowing it to use any solver defined as an instance of `float_solver`.

Other important subclasses of the `solver` type class are `herbrand` (which includes as instances all constructor types declared as `deriving solver`) and its subclass the `prolog` type class. The role of the `herbrand` type class is to distinguish between constructor types and other user defined solver types. The `prolog` class requires the type to support a number of non-logical operations commonly used in Prolog style programming. For instance, it provides `var/1` and `nonvar/1` to test if a variable is still uninstantiated or not and standard functions to access the components of a term such as `functor`. It also provides the method `===/2` which succeeds only if both its arguments are variables and constrained to be equal.

A constructor type can be declared to support the Prolog built-ins by annotating the type declaration with `deriving prolog` rather than `deriving solver`. In this case, the compiler automatically generates definitions for a `prolog` class methods as well as those for the `solver` class. Distinguishing between `herbrand` and `prolog` allows the HAL compiler to differentiate between types which are used logically from those which are not (useful for optimization).

4 Dynamic Scheduling

An important feature of the HAL language is a form of "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 when delay condition $cond_i$ is satisfied. By default, delayed goals remain active and are reexecuted whenever their delay condition becomes true again. This is useful, for example, if the delay condition is "the lower bound has changed." However, delayed goals may also contain calls to the special predicate `kill/0` which kills all delayed goals in the immediate surrounding delay construct; that is, these goals will no longer be active.

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/2` type class defined as follows:

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

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

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 more than one solver as long as they use a common delay id (the original design of delay [3] did not allow this).

The HAL compiler translates the delay construct into the base delay methods provided by the classes. Thus, the delay construct shown above is translated into:

`get_id(Id), delay(`$cond_1$`,Id,`$goal_1$`), ..., delay(`$cond_n$`,Id,`$goal_n$`)`

where each call to `kill/0` in a $goal_i$ is replaced by a call to `kill(Id)`.

Most modern logic programming languages allow predicates or goals to delay until a particular Herbrand variable is bound or is unified with another variable. In HAL a programmer can declare this by including `deriving delay` in the declaration for a constructor type. As when deriving from `solver/1` or `prolog/1`, the compiler will automatically generate the appropriate methods and instance declaration for that type. All such types use the common delay conditions `bound(X)`, `touched(X)` and the common delay id type `system_delay_id` and its system defined instance of `delay_id`. Note that `system_delay_id` can also be used in programmer defined solvers.

As an example of the use of delay in constructing constraint solvers, the following program containst the code for (part of) a simple Boolean constraint solver.[4]

[3] To simplify analysis, each $goal_i$ must be `semidet` and may not change the instantiation of variables. As a result, delayed code cannot invalidate the mode and determinism checking when woken up.

[4] Note the `touched` delayed goals are included only for illustration, they are not used in the experiments.

```
:- module bool_delay.
:- instance bool_solver(boolv).
:- export_abstract typedef boolv -> ( f ; t ) deriving [prolog,delay].
:- export func true --> boolv.
true --> t.
:- export pred and(boolv,boolv,boolv).
:-        mode and(oo,oo,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 func false --> boolv.
false --> f.
:- pred notboth(boolv,boolv).
:- mode notboth(oo,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)).
```

The constructor type `boolv` is used to represent Booleans. Notice how the class functions `true` and `false` are simply defined to return the appropriate value, while the `and` predicate delays until one argument has a fixed value, and then constrains the other arguments appropriately. In the case of `notboth` we also test if two variables are identical. Hence, `boolv` must be declared as an instance of both the `prolog` type class and the `delay` type class (and, hence, implicitly as an instance of the `solver` type class.)

5 Using External Solvers from HAL

One of the main design requirements on the HAL language is that it should readily support integration into foreign language applications and, in particular, allow constraint solvers written in other languages to be called with relatively little overhead. An example of such a solver is CPLEX [10], a simplex based solver supporting linear arithmetic constraints.[5] This section details our experience integrating CPLEX into HAL.

The HAL interface for CPLEX is built on top of three Mercury predicates: the function `initialise_cplex` which returns a CPLEX solver instance CP, the predicate `add_column(`CP`,`n`)` which adds n columns to the tableau, and the predicate `add_equality(`CP`, [(`c_1`,`v_1`),...,(`c_n`,`v_n`)], `b`)` which adds the equation $c_1 \cdot v_1 + \cdots + c_n \cdot v_n = b$ to the tableau. These predicates wrap the C interface functions of CPLEX. This is easy to do since C code can be directly written as part of a Mercury predicate body. These predicates also handle trailing and restoration of choice points. This is done by using the higher-order predicate

[5] CPLEX also provides routines for mixed integer programming and barrier methods but we have not yet integrated these.

trail/1 which places its argument (a predicate closure), on the function trail to be called in the event of backtracking to trail/1.

A naive way to write the interface in HAL is as follows.

```
:- module cplex.
:- instance linfloat_solver(cfloat).
:- import int.
:- export_abstract typedef cfloat -> col(int).
:- reinst_old cfloat = ground.
:- glob_var CPLEX has_type cplex_instance init_value initialise_cplex.
:- glob_var VarNum has_type int init_value 0.
:- export_only pred init(cfloat).
:-             mode init(no) is det.
init(V) :- V = col($VarNum), $VarNum := $VarNum + 1, add_column($CPLEX,1).
:- export_only pred cfloat = cfloat.
:-             mode oo = oo is semidet.
V1 = V2 :- add_equality($CPLEX, [(1.0,V1),(-1.0,V2)], 0.0).
:- export func cfloat + cfloat --> cfloat.
V1 + V2 --> V3 :-
      init(V3),
      trust_det add_equality($CPLEX, [(1.0,V1),(1.0,V2),(-1.0,V3)], 0.0).
:- export func float x cfloat --> cfloat.
C x V1 --> V2 :-
      init(V2),
      trustdet add_equality($CPLEX, [(C,V1),(-1.0,V2)], 0.0).
:- coerce coerce_float(float) --> cfloat.
:- export func coerce_float(float) --> cfloat.
coerce_float(C) --> V :-
      init(V),
      trust_det add_equality($CPLEX, [(1.0,V)], C).
```

The solver type `cfloat` is a wrapped integer giving the column number of the variable in the CPLEX tableau. It is exported abstractly to provide an abstract data type, and declared to be an instance of the linear arithmetic constraint solver class `linfloat_solver`. The `reinst_old` declaration states that the instantiation `old` for `cfloat`s must be interpreted as `ground` inside this module reflecting their internal implementation. We use two global variables: `CPLEX` for storing the CPLEX instance, and `VarNum` for storing the number of variables (columns) in the solver.

The predicate `init/1` simply increments the counter `VarNum` and adds a column to the CPLEX tableau. The `=/2` predicate adds an equality to the CPLEX tableau. Both are designated as `export_only`, which makes them visible outside the module, but not inside. This avoids confusion with the internal view of `cfloat`s as wrapped integers rather than the external view as float variables. The function `+/2` initialises a new variable to be the result of the addition and adds an equality constraint to compute the result. The `trust_det` annotation allows the compiler to pass the determinism check (the solver author knows that this call to `add_equality` will not fail). The linear multiplication function `x/2` is defined similarly to `+/2`.

Since `cfloats` are constrained floats, it is convenient to be able to use floating point constants in place of `cfloats`. HAL allows the solver programmer to specify the automatic coercion of a base type to a solver type. In our example, the `coerce` directive declares that `coerce_float` is a coercion function and the next three lines give its type, mode and definition.

Unfortunately, this naive interface has a high overhead. One issue is that many arithmetic constraints are simple assignments or tests which do not require the power of a linear constraint solver. Thus, we can improve the interface by only passing "real" constraints to the solver and "solving" simple assignments and tests in the interface functions themselves. This can be done easily by redefining the `cfloat` type to wrap either a true variable or a constant value and redefining our interface functions to handle the different cases appropriately.

Another issue is that the interface splits complex linear equations into a large number of intermediate constraints and variables. A better approach is to have `+` and `x` build up a data structure representing the linear constraint. More precisely, we can redefine `cfloat` to be this data structure and for `+/2`, `x/2`, `init/1` and `coerce_float/1` to build the data structure. As a by product, this data structure can also be used to track constants and perform tests and assignments in the interface. The modified code is:

```
:- export_abstract typedef cfloat -> cfloat(float,list(cterm)).
:-                 typedef cterm  -> (float,int).
init(V) :- V = cfloat(0.0,[(1.0,$VarNum)]),
           $VarNum := $VarNum + 1, add_column($CPLEX,1).
cfloat(C1, Vs1) = cfloat(C2, Vs2) :-
           negate_coeffs(Vs2, NewVs2),
           append(Vs1, NewVs2, Terms),
           add_equality($CPLEX, Terms, C2-C1).
cfloat(C1, Vs1) + cfloat(C2, Vs2) -->
           cfloat(C1+C2,Vs) :- append(Vs1, Vs2, Vs).
C x cfloat(F, Vs) --> cfloat(C*F,NewVs) :- multiply_coeffs(C, Vs, NewVs).
coerce_float(C) --> cfloat(C, []).
```

Also, in external solvers such as CPLEX that are not specialized for incremental satisfiability checking, the usual CLP approach of checking satisfiability after each new constraint is added, may be expensive. We can therefore improve performance by "batching" constraints and requiring the programmer to explicitly call the solver to check for satisfiability.

6 Using Constraint Handling Rules (CHRs)

Constraint Handling Rules (CHRs) have proven to be a very flexible formalism for writing incremental constraint solvers and other reactive systems. In effect, the rules define transitions from one constraint set to an equivalent constraint set. Rules are repeatedly applied until no new rule can be applied. Once applied, a rule cannot be undone. For more details the interested reader is referred to [6].

The simplest kind of rule is a *propagation* rule of the form

$$lhs \texttt{ ==> } guard \texttt{ | } rhs$$

where *lhs* is a conjunction of CHR constraints, *guard* is a conjunction of constraints of the underlying language (in practice this is any goal not involving CHR constraints) and *rhs* is a conjunction of CHR constraints and constraints of the underlying language. The rule states that if there is a set S appearing in the global CHR constraint store G that matches *lhs* such that goal *guard* is entailed by the current constraints, then we should add the *rhs* to the store. *Simplification rules* have a similar form (replacing the `==>` with a `<=>`) and behavior except that the matching set S is deleted from G. A syntactic extension allows only part of the *lhs* to be eliminated by a simplification rule:

$$lhs_1 \setminus lhs_2 \texttt{ <=> } guard \texttt{ | } rhs$$

indicates that only the set matching lhs_2 is eliminated.

Efficient implementations of CHRs are provided for SICStus Prolog, Eclipse Prolog (see [5]) and Java [11]. Recently, they have also been integrated into HAL [8]. As in most implementations, HAL CHRs sit on top of the "host" language. More exactly, they may contain HAL code and are essentially compiled into HAL in a pre-processing stage of the HAL compiler. As a consequence, CHR constraints defined in HAL require the programmer to provide type, mode and determinism declarations.

The following program implements part of a Boolean solver implemented in HAL using CHRs.[6]

```
:- module bool_chr.
:- instance bool_solver(boolv).
:- export_abstract
   typedef boolv -> wrap(int).
:- reinst_old boolv = ground.

:- glob_var VNum
   has_type int init_value 0.

:- export_only
   pred init(boolv).
:- mode init(no) is det.
init(V) :- V = wrap($VNum),
           $VNum := $VNum + 1.

:- export constraint true(boolv).
:-        mode true(oo) is semidet.
:- export constraint false(boolv).
:-        mode false(oo) is semidet.
true(X), false(X) <=> fail.
:- export constraint
          and(boolv,boolv,boolv).
:-        mode and(oo,oo,oo) is semidet.
true(X) \ and(X,Y,Z) <=> Y = Z.
true(Y) \ and(X,Y,Z) <=> X = Z.
false(X) \ and(X,Y,Z) <=> false(Z).
false(Y) \ and(X,Y,Z) <=> false(Z).
false(Z) \ and(X,Y,Z) <=> notboth(X,Y).
true(Z) \ and(X,Y,Z) <=> true(X), true(Y).
```

In this case `boolv`s are simply variable indices[7] and Boolean constraints and values are implemented using CHR constraints. Initialization simply builds a new term and increments the Boolean variable counter `VNum` which is a global variable. The `constraint` declaration is like a `pred` declaration except it indicates that it is a CHR predicate. The mode and determinism for each CHR constraint are defined as usual. The remaining parts are CHRs. The first rule

[6] Somewhat simplified for ease of exposition.
[7] HAL does not yet support CHRs on Herbrand types

states that if a variable is given both truth values, true and false, we should fail. The next rule (for `and/3`) states that if the first argument is true we can replace the constraint by an equality of the remaining arguments.
In HAL, CHR constraints must have a mode which does not change the instantiation of their arguments (like `oo` or `in`) to preserve mode safety, since the compiler is unlikely to statically determine when rules fire. Predicates appearing in the guard must also be `det` or `semidet` and not alter the instantiation of variables appearing in the left hand side of the CHR (this means they are implied by the store). This is a weak restriction since, typically, guards are simple tests.

7 Evaluation

Our first experiment has three aims. First, it illustrates the use of type classes for "plug and play" with solvers. Second, it determines the overhead of using type classes when implementing solvers. Third, it evaluates the efficiency of the generic solver writing constructs supported by HAL: dynamic scheduling and CHRs. For this experiment we created three implementations of a propagation-based Boolean constraint solver: using dynamic scheduling (*dyn*); using CHRs (*chr*); and using conversion to integer constraints (*int*).[8] We give two results for each HAL solver: $solv_t$ which uses type classes for separate compilation, where each query module was compiled separately from the solver, and joined at link time; and $solv_i$ where the query module imported the solver, and was compiled with this knowledge, so removing the overhead of type classes. It is important to note that type classes allowed us to use identical code for the benchmarks: only at linking time did we need to choose which solver to use.
To evaluate the efficiency of HAL,[9] we also built comparable solvers in SICStus Prolog: using the generic `when` delay mechanism ($SICS_w$) closest to our generic delay mechanism, using the CHRs of SICStus ($SICS_c$); and using the `clfd` integer propagation solver ($SICS_z$). Finally, for interest, we provide two more SICStus solvers: ($SICS_b$) a dynamic scheduling solver using the highly restricted but efficient `block` mechanism of SICStus, and ($SICS_v$) where the ground variable numbers in the CHR solver are replaced by Prolog variables, allowing the use of attribute variable indices.
The comparison uses five simple Boolean benchmarks (most from [2]): the first `pigeon`n-m places n pigeons in m pigeon holes (the 24-24 query succeeds, while 8-7 fails); `schur`n Schurs's lemma for n (see [2]) (the 13 query is the largest n that succeeds); `queens`n the Boolean version of this classic problem; `mycie`n-m which colors a 5-colorable graph (taken from [16]) with n nodes and

[8] More exactly, we use the integer propagation solver described in [7] which is implemented in C and interfaced to HAL using the methodology described in Section 5.
[9] It is much easier to build highly flexible but inefficient mechanisms for defining solvers.

Benchmark	Var	Con	Search	Dynamic Scheduling			
				dyn_t	dyn_i	$SICS_w$	$SICS_b$
mycie23_71	184	583	19717	1855	1816	34769	1920
fulladder	135	413	1046	472	471	5181	147
pigeon24_24	1152	13896	576	805	822	2258	56
pigeon8_7	112	444	24296	931	901	16870	843
queens18	972	13440	42168	8904	8818	125250	7316
schur13	178	456	57	22	18	118	4
schur14	203	525	450	98	112	1308	63

Table 1. Comparison of Boolean solvers for dynamic scheduling.

Benchmark	CHRs				Integer		
	chr_t	chr_i	$SICS_c$	$SICS_v$	int_t	int_i	$SICS_z$
mycie23_71	25073	25070	200613	76567	1279	1251	5339
fulladder	4240	4270	24770	13840	178	175	313
pigeon24_24	71455	70785	107366	71048	81	72	957
pigeon8_7	11313	11176	61126	36251	765	726	3166
queens18	504750	511620	1350636	263433	4522	4438	12363
schur13	53	63	450	201	9	7	51
schur14	823	830	5966	2319	55	52	278

Table 2. Comparison of Boolean solvers using an existing integer solver and CHRs.

m edges with 4 colours; and `fulladder` which searches for a single faulty gate in a n bit adder (see e.g. [14] for the case of 1 bit).

Table 1 gives an indication of how much work the solvers are performing for each benchmark. Var is the number of variables initialised by the solver, Con is the number of Boolean constraints, and Search is the number of labeling steps performed (using the default labeling strategy to find a first solution). Note that each solver implements *exactly* the same propagation strength on Boolean constraints and, thus, for each benchmark each different solver performs exactly the same search. All timings are the average over 10 runs on a dual Pentium II-400MHz with 384M of RAM running under Linux RedHat 5.2 with kernel version 2.2, and are given in milliseconds. SICStus Prolog 3.8.4 is run under compact code (no fastcode for Linux).

From Table 1 it is clear that the generic delay mechanism implemented in HAL is reasonably efficient. In comparison with the propagation happening in C in the integer solver, the dynamic scheduled version is only 4 times slower. It also compares well with the generic dynamic scheduling of SICStus. However, the `block` based dynamic scheduling of SICStus illustrates how delay that is tightly tied to the execution mechanism can be very efficient.

Table 2 shows that the CHR solver mechanism for HAL (at least for this example) is significantly faster than the SICStus equivalent, so much so that in this case even the use of attributed variable indexing does not regain the

Bench	*naive*		*constants*			*datastructures*				*ECL*	
	Con	*inc*	Con	*inc*	*batch*	Con	*inc*	*batch*	*+opt*	Con	*batch*
`fib`	2557	1329000	465	49010	5950	233	25280	2910	2690	232	7650
`laplace`	347	9070	298	5140	1550	20	950	210	210	90	1070
`matmul`	nonlinear		684	142830	15020	216	27390	2950	2270	432	10050
`mortgage`	nonlinear		482	89940	18200	2	810	750	1520	240	6390

Table 3. Executing CPLEX using the various HAL interfaces.

difference except in the biggest examples.[10] We are currently working on adding indices to HAL CHRs.
Examination of both tables shows that the type class mechanism does not add substantial overhead to the use of constraint solvers: The overhead of type classes varies up to 3.5% (ignoring the 28% on very small times), and the average overhead is just 2%.
Note that this experiment is not meant to be an indication of the merits of the different approaches since, for building different solvers, each approach has its place.
Our second experiment compares the speed of the HAL interfaces defined in Section 5: the *naive* interface, the interface *constant* that keeps track of when `cfloat`s are constants and solves assignments and test in the interface itself, and the interface *datastructures* which builds data structures to handle functions calls, and only sends constraints at predicate calls. For the last two, we run the solver incrementally (solving after every constraint addition) and in batch mode (explicitly calling a `solve` predicate). For the last interface we also provide a version (`+opt`) which implements a simple type of partial evaluation by making use of Mercury to aggresively inline predicates and functions even across module boundaries. Finally, we compare against the ECLiPSe [9] interface (*ECL*) to the CPLEX solver, which also batches constraints.
The benchmarks are standard small linear arithmetic examples (see e.g. [3]). The table gives the number of constraints sent to CPLEX by each solver (Con), and execution times in milliseconds for 100 executions of the program. The last two benchmarks involve nonlinear constraints (not handled by CPLEX) if constants are not kept track of.
It is clear from Table 3 that the naive interface is impractical. Tracking constants and performing assignments and tests in the interface itself significantly improves speed. The move to using data structures to build linear expressions is clearly important in practice. Using this technique, the number of constraints passed to CPLEX for `mortgage` is reduced to just 2. Finally, batching is clearly worthwhile in these examples.
This experiment shows that the external solver interface for CPLEX is considerably faster than that provided by ECLiPSe and we believe that there is still considerable scope for improvement. Inlining of predicates and functions

[10] Note that using bindings to represent `true` and `false` would result in a more efficient SICStus CHR Boolean solver, but the equivalent is not possible in HAL (at present).

across module boundaries provides substantial improvement, but we believe that we can do even better by partially evaluating away many of the calls to solver interface and building the arguments to the calls to CPLEX at compile time if the constraint is known. Similarly, we would like to automatically perform "batching" by making HAL introduce satisfiability checks just before a choice point is created. An important lesson from the second experiment is that it is vital for a CLP language to allow easy experimentation with the interface to external solvers, since the choice of interface can make a crucial difference to performance. Our experience with HAL has been very positive in this regard.

References

1. F. Bueno, M. Garcia de la Banda, M. Hermenegildo, K. Marriott, G. Puebla, and P.J. Stuckey. A model for inter-module analysis and optimizing compilation. In *Procs of LOPSTR2000*, volume 2042 of *LNCS*, pages 86–102, 2001.
2. P. Codognet and D. Diaz. Boolean constraint solving using `clp(FD)`. In *Procs. of ILPS'1993*, pages 525–539. MIT Press, 1993.
3. B. Demoen, M. García de la Banda, W. Harvey, K. Marriott, and P.J. Stuckey. An overview of HAL. In *Procs. of PPCP'99*, LNCS, pages 174–188, 1999.
4. A.J. Fernández and B.C. Ruiz Jiménez. Una semántica operacional para CProlog. In *Proceedings of II Jornadas de Informática*, pages 21–30, 1996.
5. T. Frühwirth. CHR home page. www.informatik.uni-muenchen.de/~fruehwir/chr/.
6. T. Frühwirth. Theory and practice of constraint handling rules. *Journal of Logic Programming*, 37:95–138, 1998.
7. W. Harvey and P.J. Stuckey. Constraint representation for propagation. In *Procs. of PPCP'98*, LNCS, pages 235–249. Springer-Verlag, 1998.
8. C. Holzbaur, P.J. Stuckey, M. García de la Banda, and D. Jeffery. Optimizing compilation of constraint handling rules. In *Procs. of ICLP17*, LNCS, 2001.
9. IC PARC. ECLiPSe prolog home page. http://www.icparc.ic.ac.uk/eclipse/.
10. ILOG. CPLEX product page. http://www.ilog.com/products/cplex/.
11. Jack: Java constraint kit. http://www.fast.de/ mandel/jack/.
12. D. Jeffery, F. Henderson, and Z. Somogyi. Type classes in Mercury. Technical Report 98/13, University of Melbourne, Australia, 1998.
13. S. Kaes. Parametric overloading in polymorphic programming languages. In *ESOP'88 Programming Languages and Systems*, volume 300 of *LNCS*, pages 131–141, 1988.
14. K. Marriott and P.J. Stuckey. *Programming with Constraints: an Introduction*. MIT Press, 1998.
15. Z. Somogyi, F. Henderson, and T. Conway. The execution algorithm of Mercury: an efficient purely declarative logic programming language. *JLP*, 29:17–64, 1996.
16. M. Trick. mat.gsia.cmu.edu/COLOR/color.html.
17. P. Wadler and S. Blott. How to make *ad-hoc* polymorphism less *ad-hoc*. In *Proc. 16th ACM POPL*, pages 60–76, 1989.