To the Gates of HAL: a HAL tutorial

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

[1] School of Computer Science & Software Engineering, Monash University, Australia
[2] Dept. of Computer Science, K.U.Leuven, Belgium
[3] Dept. of Computer Science & Software Engineering, University of Melbourne, Australia

Abstract. Experience using constraint programming to solve real-life problems has shown that finding an efficient solution to the problem often requires experimentation with different constraint solvers or even building a problem-specific constraint solver. HAL is a new constraint logic programming language expressly designed to facilitate this process. It provides semi-optional type, mode and determinism declarations. These allow natural constraint specification by means of type overloading, better compile-time error checking and generation of more efficient run-time code. Importantly, it provides type classes which can be used to specify solver interfaces, allowing the constraint programmer to support modelling of a constraint problem independent of a particular solver, leading to easy "plug and play" experimentation with different solvers. Other interesting features include mutable global variables for implementing a constraint store, and dynamic scheduling and Constraint Handling Rules (CHRs) for combining, extending and writing new constraint solvers.

1 Introduction

Constraint logic programming (CLP) languages are evolving to support more flexible experimentation with constraint solvers. First generation CLP languages, such as CLP($\mathcal{R}$) [15], provided almost no support. They had a fixed underlying solver for each constraint domain which was viewed as a closed "black box." Second generation CLP languages, such as `clp(fd)` [6], provided more support by viewing the solver as a "glass box" which could be extended to support problem-specific complex constraints. However, CLP programmers want more than this: they want to be able to develop new problem-specific constraint solvers, for example by using "hybrid" methods that combine different constraint solving techniques. For this reason, recent versions of the CLP languages ECLiPSe and SICStus support the addition and specification of new constraint solvers by providing features such as dynamic scheduling, constraint handling rules [8] and attributed variables [12].

We describe the CLP language, HAL, which has been expressly designed to support experimentation with different constraint solvers and development of new solvers. Our specific design objectives were five-fold:

- *Efficiency*: Current CLP languages are considerably slower than traditional imperative languages such as C. This efficiency overhead has limited the use of CLP languages, and becomes even more of an issue when constraint solvers are to be (partially) implemented in the language itself.
- *Integrability*: It should be easy to call solvers written in other languages, e.g. C, with little overhead. Conversely, it should be possible for HAL code to be readily called from other languages, facilitating integration into larger applications. Although most CLP languages provide a foreign language interface, it is often complex and may require rewriting the foreign language code to use "safe" memory management routines.
- *Robustness*: Current CLP languages provide little compile-time checking. However, when developing complex multi-layered software such as constraint solvers compile-time checking becomes crucial for improving program robustness.
- *Flexible choice of constraint solvers:* It should be easy to "plug and play" with different constraint solvers over the same domain.
- *Easy definition of new solvers:* it should be straightforward to extend an existing solver, create a hybrid solver by combining solvers and to write a new constraint solver.

HAL has four interesting features which allow it to meet these objectives. The first is semi-optional type, mode and determinism declarations for predicates and functions. Information from the declarations allows the generation of efficient target code, improves robustness by using compile-time tests to check that solvers and other procedures are being used in the correct way, and facilitates 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. For example, "=" is overloaded to be equality on all constraint domains, and type inference determines if this is equality over terms, reals etc.

The second feature is type classes. These allow specification of an abstract interface for solvers and so facilitate "plug and play" experimentation with different solvers over the same domain.

The third feature is support for "propagators" by means of a specialized delay construct. HAL allows the programmer to annotate goals with a delay condition which tells the system that execution of that goal should be delayed until the condition is satisfied. By default, the delayed goal remains active and is reexecuted whenever the delay condition becomes true again. Such dynamic scheduling of goals is useful for writing simple constraint solvers, extending a solver and combining different solvers. Importantly, delay conditions and the method for handling delayed goals is solver dependent, but with a common interface provided as a standard type class.

The fourth feature is "global variables." These behave a little like C's static variables and are only visible within a module. They are not intended for general use; rather they allow communication between search branches, and allow solver writers to efficiently implement a persistent constraint store.

Broadly speaking, HAL unifies two recent directions in constraint programming language research. The first direction is that of earlier CLP languages, including CLP($\mathcal{R}$), `clp(fd)`, ECLiPSe and SICStus. The second direction is that of logic programming languages with declarations as exemplified by Mercury [20]. Earlier CLP languages provided constraints and constraint solvers for pre-defined constraint domains and many provided dynamic scheduling. However, they did not allow type, mode, determinism or type class declarations. Providing such declarations has influenced the entire design of HAL, from the module system to delay constructs. Another important difference is explicit language support for extending or writing constraint solvers.

Like HAL, the Mercury language also provides type, mode, determinism and type class declarations. It is probably the most similar language to HAL, and we have leveraged greatly from its sophisticated compilation support by using it as an intermediate target language. The key difference is that Mercury is logic programming based and does not support constraints and constraint solvers. Indeed, it does not even fully support Herbrand constraints since it provides only a limited form of unification.

This paper provides a high-level introduction to HAL. For more detailed explanation of various aspects of HAL the interested reader is referred to our earlier publications [4, 3, 9, 2, 13]. In the next section we introduce the HAL language. In Section 3 we discuss the declarations supported by HAL, then in Section 4 we elaborate on how constraint solvers fit within in the language. In Section 5 we illustrate the user-extensible delay mechanism provided by HAL. Next in Section 6 we describe built-in Herbrand constraint solving facilities in HAL, then in Section 7 we examine the built-in solver hierarchy. Section 8 discusses how to build constraint solvers in HAL. In Section 9 we discuss the current implementation and Section 10 concludes with a discussion of future work.

2 A First Example

The basic HAL syntax follows the standard CLP syntax, with variables, literals, rules and predicates defined as usual (see, e.g., [18] for an introduction to CLP). Our philosophy has been to design a language which is as pure as possible, without unduly compromising efficiency. The module system in HAL is similar to that of Mercury. A module is defined in a file, it `import`s the modules it uses and has `export` annotations on the declarations for 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` for modelling the relationship between P the principal or amount owed, T the number of periods in the mortgage, I the interest rate of the mortgage, R the repayment due each period of the mortgage and B the balance owing at the end.

```
:- module mortgage.                                                (L1)
:- export pred mortgage(CF,CF,CF,CF,CF) <= float_solver(CF).       (L2)
:-        mode mortgage(in,in,in,in,out) is nondet.                (L3)
:-        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, NP = P + P * I - R,               (R2)
                       mortgage(NP,T-1.0,I,R,B).
```

The first line ($L1$) states that this is the definition of the module `mortgage`. Line ($L2$) declares that this module exports the predicate `mortgage` which is polymorphic in the type of its as argument but the type must be an instance of the `float_solver` type class. This is the *type* declaration for `mortgage`. Lines ($L3$) and ($L4$) are examples of *mode of usage* declarations. Since there are two declarations, `mortgage` has two possible modes of usage. In the first, the first four arguments have an `in` mode meaning their values are fixed when the predicate is called, and the last has a mode `out` which means it is uninitialized when called, and fixed on the return from the call to `mortgage`. Line ($L4$) gives another mode for the `mortgage` where each argument has mode `oo` meaning that each argument takes a "constrained" variable and returns a "constrained" variable. This is a more flexible mode of usage but will be less efficient to execute. The two declarations also state that for either mode `mortgage` is `nondet` meaning that the query may return 0 or more answers. Actually, in the case of ($L3$) it would be more precise to declare it as `semidet`, meaning that it either fails (for example `mortgage(0.0,-1.0,0.0,0.0,B)` fails) or succeeds with exactly one answer. However, HAL is currently unable to confirm it since this would require reasoning over numbers.

The rest of the file contains the standard two rules defining `mortgage`. The first states that when the number of repayments is 0, then the balance is simply the principal. The second states that if the number of repayments is greater than one, then we make one repayment, compute the new principle `NP` and take out a mortgage with this new principle for one less time period.

Since the definition of `mortgage` is polymorphic we can use it with any constraint solver for `floats`. In the following code we use the standard `clpr` solver, which provides a simplex-based arithmetic constraint solver for constrained floats, called `cfloats`. It is based on the CLP($\mathcal{R}$) solver.

```
:- module main.
:- import mortgage,clpr,io.
:- export io impure pred main.
:-                mode main is cc_multi.
main :- ( mortgage(10000,10,0.10,1500,B), V = val(B) ->
          io write("The balance remaining is:"),
          io write(V)
        ; io write("No solution")).
```

The predicate `main` is declared to be `io` to indicate that it makes use of the I/O routines and hence an I/O state is implicitly threaded through the code.

It is also declared to be `impure` since it uses the impure `val` function. The `val` function returns the value of a solver variable if it is has a fixed value and fails otherwise. The determinism `cc_multi` means that `main` succeeds at least once, but we are interested in only the first solution.

3 Declarations

As we can see from the above example, one of the key features of HAL is that programmers may annotate predicate definitions with declarations. Information from the declarations allows the generation of efficient target code, compile-time tests to check that solvers and other predicates are being used in the correct way and facilitates integration with foreign language procedures. Currently the compiler supports type, mode, determinism, I/O and purity declarations, but the aim is to eventually support general user-defined declarations similar to those supported by the CIAO language [11].

By default, declarations are checked at compile-time, generating an error if they cannot be confirmed by the compiler. However, the programmer can also provide "trust me" declarations. These generate an error if the compiler can definitely prove they are wrong, but otherwise the compiler trusts the programmer and generates code according to the trusted declarations. A compile-time warning is issued if the declaration cannot be confirmed by the compiler.

Type declarations: These specify the representation format of a variable or argument. Thus, for example, the type system distinguishes between constrained floats provided by the `clpr` solver and the standard numerical float since these have a different representation. Types are specified using type definition statements. They are (polymorphic) regular tree type statements. For instance, lists are defined as:

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

Equivalence types are also supported. For example,

```
:- typedef vector = list(float).
```

Ad-hoc overloading of predicates is also allowed, although the predicates for different type signatures must be in different modules.

As an example, imagine that we wish to write a module for handling complex numbers. We can do this by leveraging from the `clpr` solver defining `cfloats`.

```
:- module complex.
:- import clpr.
:- export_abstract typedef complex -> c(cfloat,cfloat).
:- export pred cx(cfloat,cfloat,complex).       % access/creation
:-        mode cx(in,in,out) is det.
:-        mode cx(out,out,in) is det.
:-        mode cx(oo,oo,oo) is semidet.
cx(X,Y,c(X,Y)).
:- export func complex + complex --> complex.   % addition
:-        mode in + in --> out is det.
:-        mode oo + oo --> oo is semidet.
c(X1,Y1) + c(X2,Y2) --> c(X1+X2,Y1+Y2).
```

Note that the type definition for `complex` is exported abstractly, which means that the internal representation of a complex number is hidden within the module. This ensures that code cannot create or modify complex numbers outside of the `complex` module. Thus, this module also needs to export a predicate, `cx`, for accessing and creating a complex number. As this example demonstrates, HAL also allows the programmer to declare functions. The symbol "`-->`" should be read as "returns."

Using this module the programmer can now use complex arithmetic as if it were built into the language itself. If both `clpr` and `complex` are imported, type inference will determine the type of the arguments of each call to `+` and appropriately qualify the call with the correct module.

Mode declarations: These specify how execution of a predicate modifies the "instantiation state" of its arguments. A mode is associated with a predicate argument and has the form $Inst_1 \rightarrow Inst_2$, where $Inst_1$ and $Inst_2$ describe the input and output instantiation states of the argument, respectively.

The *base* instantiation states 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 has a known 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 ($L3$) in our example program declares that each argument of `mortgage` has mode `oo`, i.e., takes an `old` variable and returns an `old` variable.

More sophisticated instantiation states (lying between `old` and `ground`) may be used to describe the state of complex terms. Instantiation state definitions look something 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 state.

Mode definitions have the following syntax:

```
:- modedef to_groundlist -> (fixed_length_list -> ground).
:- modedef same(I) -> (I -> I).
```

We have already seen examples of predicate mode declarations in the previous two programs. As another example, a mode declaration for an integer variable labelling predicate `labeling` would be

```
:- mode labeling(to_groundlist) is nondet.
```

Mode checking is a relatively complex operation involving reordering body literals in order to satisfy mode constraints, and inserting initialization predicates for solver variables. The compiler performs multi-variant specialization by generating different code for each declared mode for a predicate. The code corresponding to a mode is referred to as a "procedure" and calls to the original predicate are replaced by calls to the appropriate procedure.

Determinism declarations: These detail how many answers a predicate may have. We use the Mercury hierarchy: `nondet` means any number of solutions; `multi` at least one solution; `semidet` at most one solution; `det` exactly one solution; `failure` no solutions; and `erroneous` a runtime error. In addition the determinisms `cc_multi` and `cc_nondet` correspond to `multi` and `nondet` in a context where we are only interested in the first solution.

I/O declarations: Predicates and literals that use I/O safely must be annotated with `io`. The compiler will then automatically thread two extra arguments holding the I/O state before and after execution. Predicates can also use I/O unsafely by annotating calls to I/O literals with `unsafe_io`. The compiler will then build a dummy input I/O state for the literal and discard the resulting I/O state. This is useful, for example, for debug printing. I/O predicates are required to be `det` or `cc_multi`.

Purity declarations: These capture whether a predicate is `impure` (affects or is affected by the computation state), or `pure` (otherwise). By default predicates are pure. Any predicate that uses an impure predicate must have its predicate declaration annotated as either `impure` (so it is also impure) or `trust pure` (so even though it uses impure predicates it is considered pure). Note that all constraints are in some sense impure since they change the constraint store. However, for a correct solver, they act and should be treated as logically pure.

4 Constraint Solvers and Type Classes

As we have seen HAL provides type classes [17, 21]. These 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 [16] and CProlog [7] were the first logic programming languages to include them. One major motivation for providing type classes in HAL is that they provide a natural way of specifying a constraint solver's interface and separating this from its implementation 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 current module must either define the method `=/2` with type `int=int` and mode `oo=oo is semidet` or indicate in the instance definition that the method is a renaming of another predicate.

Like Mercury, all types in HAL have an associated "equality" for modes `in=out` and `out=in`, which correspond to assignment, construction or deconstruction. These are implemented using specialised procedures rather than the `=/2` method. In addition, most types support testing for equality, the main exception being for types with higher-order subtypes. Thus, HAL automatically generates instances of `eq` for all constructor types which do not contain higher-order subtypes and for which the programmer has not already declared an instance.

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

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

Note the use of the abbreviated syntax for a single combined type, mode and determinism declaration. The above declaration 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 methods `init/1` and `=/2` for this type with mode and determinism as shown.

A *solver type* is a type which is an instance of `solver/1`. The compiler distinguishes between variables of a solver type (*solver variables*) and other during mode analysis. The distinction first determines the interpretation of the instantiation `old`. Only solver variables can be truly `old`, for other variables `old` is interpreted as bound. Second, when necessary the compiler automatically inserts calls to `init` in order to change the instantiation of solver variables from `new` to `old`. See [9] for more details.

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 of usage but not in others. In the case of `eq/1`, this will be true if equalities are found to be simple assignments or deconstructions in some modes of usage 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.[1]

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 below 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].
:- pred append_1(list(T)::in,list(T)::in,list(T)::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].
:- pred append_2(list(T)::in,list(T)::out,list(T)::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).
```

Any procedure calling `append_2` will also inherit the `eq(T)` class constraint.

5 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

$cond_1$ `==>` $goal_1$ `|` $\cdots$ `|` $cond_n$ `==>` $goal_n$

where the goal $goal_i$ will be executed when delay condition $cond_i$ is satisfied. By default, delayed goals remain active and are reexecuted every time the delay condition becomes true. This is useful, for example, if the delay condition is "the lower bound has changed." Delayed goals may also contain calls to the special predicate `kill/0`. When this is executed, all delayed goals in the immediate surrounding delay construct are killed; that is, will never be executed again.

For example, assume the delay conditions `lbc(`V`)`, `ubc(`V`)` and `fixed(`V`)` are respectively satisfied when the lower bound changes for variable V, the upper bound changes for V, and V is given a fixed value. Assume also the given functions `lb`, `ub` and `val` respectively return the current lower bound, upper bound

[1] 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].

and value of their argument, while the predicates upd_lb, upd_ub and upd_val update these bounds. Then, the following delay construct implements bounds propagation for the constraint $X \leq Y$:

```
lbc(X) ==> upd_lb(Y,lb(X)) | fixed(X) ==> upd_lb(Y,val(X)), kill |
ubc(Y) ==> upd_ub(X,ub(Y)) | fixed(Y) ==> upd_ub(X,val(Y)), kill
```

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

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

where type I represents the unique identifier (id) of each delay construct, type D represents the supported delay conditions, delay/3 takes a delay condition, an id and a goal,[2] and stores the information in order to execute the goal whenever the delay condition holds, get_id/1 returns an unused id, and kill/1 causes all goals delayed for the input id to no longer wake up.

The HAL compiler translates 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). The separation of the delay type class into two parts allows different solver types to share delay ids. Thus, we can build delay constructs which involve conditions belonging to more than one solver as long as they use a common delay id.

Method delay/3 is an example of a higher-order predicate. Like Mercury, HAL supports higher-order programming through construction of higher-order objects and higher-order calls. For example, map/3 can be written as

```
:- export pred map(list(X), pred(X, Y), list(Y)).
:-        mode map(in, pred(in, out) is det, out) is det.
:-        mode map(in, pred(in, out) is semidet, out) is semidet.
map([],_,[]).
map([X | Xs], Pred, [Y | Ys]) :- call(Pred, X, Y), map(Xs, Pred, Ys).
```

Note the higher-order type pred(X,Y) and corresponding higher-order instantiations pred(in, out) is det and pred(in, out) is semidet which combine the mode and instantiation information of the higher-order argument.

HAL is designed to make it easy to combine constraint solvers into new hybrid constraint solvers. The delay construct is an important tool for doing this. Imagine combining an existing propagation solver (variable type cint in module bounds) and an integer linear programming solver (type ilpint in module

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

cplex) to create a combined solver (type combint). Each variable in the combined solver is a pair of variables, one from each of the underlying solvers. Constraints in the combined solver create corresponding constraints for both underlying solvers. Communication between the solvers is managed by goals delayed when a variable is initialized. A sketch of such a module (with communication only from the propagation solver to the ILP solver) is given below.

```
:- module combined.
:- import bounds, cplex.
:- export_abstract typedef combint -> p(cint,ilpint).
:- export pred combint >= combint.
:-        mode oo >= oo is semidet.
p(XB,XC) >= p(YB,YC) :- XB >= YB, XC >= YC.
:- export pred init(combint::no) is det.
init(p(XB,XC)) :- init(XB), init(XC), (communicate(XB,XC) -> true ; error).
:- trust pure pred communicate(cint::oo,ilpint::oo) is semidet.
communicate(XB,XC):-
          ( lbc(XB) ==> XC >= lb(XB)
          | ubc(XB) ==> ub(XB) >= XC
          | fixed(XB) ==> XC = val(XB), kill).
```

Note how the initialization sets up the delaying communiciation goal. Since its determinism is semidet it needs to be wrapped in an if-then-else to pass determinism checking. Similarly the use of impure functions lb, ub and val require the trust pure declaration.

6 Herbrand Constraint Solving

Most HAL types are structured data types defined using constructors. For example, earlier we defined the (polymorphic) list/1 type using the constructors [] (nil) and "." (cons). Elements of these type are the usual list terms. As indicated previously, the HAL base language only provides limited operations for dealing with such data structures unless their type is an instance of the solver/1 class.

The herbrand/1 type class captures those constructor types that are solver types. An instance of the herbrand/1 type class is created by annotating the type definition with deriving solver. The compiler will then automatically generate appropriate instances for the herbrand/1, solver/1 and eq/1 classes (including the =/2 and init/1 predicates). Thus, the type declaration given earlier for lists

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

defines Mercury lists which have a fixed length while

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

defines true "Herbrand" lists.

The herbrand/1 type class supports a number of other logical and non-logical operations commonly used in Prolog style programming. For instance, it provides impure methods var/1 and nonvar/1 to test if a variable is uninstantiated or not. It also provides the impure method ===/2 which succeeds only if its arguments are both variables and are identical.

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 allow this by using the notation `deriving delay` when defining a type. The compiler will then automatically generate an instance of the `delay/2` class in addition to those of `herbrand/1`, `solver/1`, and `eq/1` classes for that type. All Herbrand types use the common delay conditions `bound(X)` and `touched(X)`, 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 by programmer defined solvers.

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

Fig. 1. Partial Boolean solver implemented by dynamic scheduling.

As an example of the use of delay in constructing solvers, Figure 1 contains the code for (part of) a simple Boolean constraint solver. The constructor type `boolv` is used to represent Booleans. Notice how the `and/3` predicate delays until one argument has a fixed value, and then constrains the other arguments appropriately. In the case of predicate `notboth/2` we also test if two variables are identical, thus using an `impure` predicate. However, since the actions of `notboth` from outside are pure, we need to use a `trust pure` declaration.

7 The Solver Hierarchy

HAL is intended to provide a wide variety of different constraint solvers. It provides a rich collection of pre-defined type classes to allow the programmer to precisely describe the interface of a particular solver. One of the most important built-in type classes is `solver_for`. which connects a base type `B` with the corresponding solver type `S`:

```
solver_for(S,B) <= solver(S) where [
    func coerce(B::in) --> S::no is det,
    impure func val(S:oo) --> B::out is semidet,
]
```

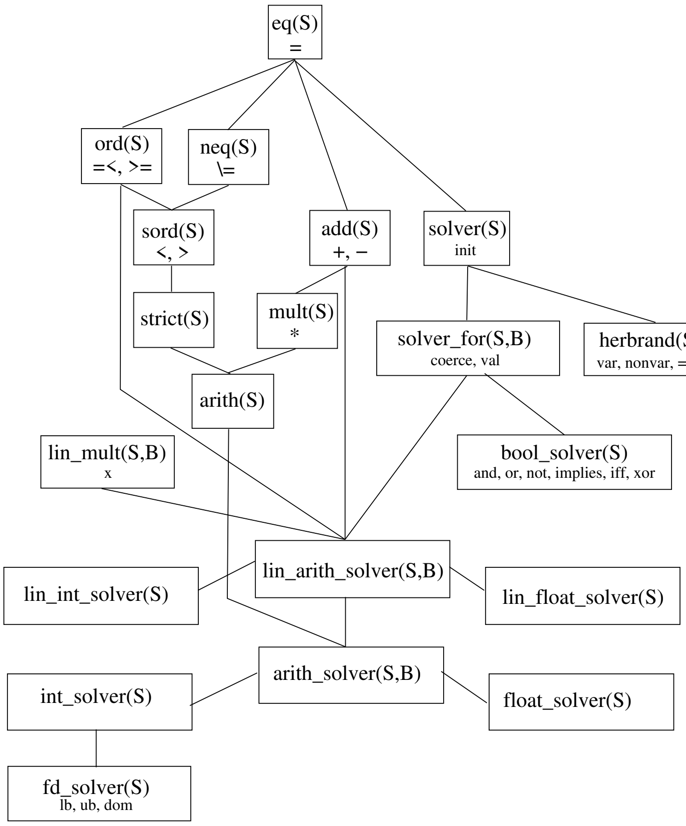

Fig. 2. HAL solver type class hierarchy.

It provides methods to “coerce” an element of the base type to that of the solver type and to find the value of a variable.

Thus, we might use the following code to declare a restricted Boolean solver

```
:- export class my_bool_solv(BV) <= solver_for(BV,bool) where [
         and(BV::oo,BV::oo,BV::oo) is semidet,
         notboth(BV::oo,BV::oo,BV::oo) is semidet ].
```

We can then declare our previous Boolean solver to be an instance of this class and provide the coercion function and `val` functions

```
:- export func coerce(bool::in) --> boolv::no is det.
coerce(true) --> t.
coerce(false) --> f.
:- export impure func val(boolv:oo) --> bool:out is semidet.
val(X) --> B :- nonvar(X), (X = t, B = true ; X = f, B = false).
```

HAL provides a hierarchy of pre-defined type classes for common constraint domains which derive from the `solver` type class: `bool_solver`, `lin_float_solver`, `float_solver`, `lin_int_solver`, `int_solver`, and `fd_solver`. The solver type classes provide a standard interface to solvers which facilitates "plug and play" experimentation. The current[3] solver type class hierarchy is illustrated in Figure 2. The new predicates and functions are listed for each class, the modes for predicates have `oo` arguments and are `semidet`, while functions have their default mode: `oo` for inputs, `no` for outputs and are `det`. The exceptions are `init`, `coerce`, `val`, `lb`, `ub` defined previously, the linear multiplication function `x` which have type `B x S --> S`, and the impure function `dom` which returns the list of integers in a finite domain variables current domain.

8 Writing Constraint Solvers in HAL

As we have indicated, HAL is designed to allow the programmer to write their own constraint solver. This section discusses a number of additional features provided by HAL to support this.

One complication of writing a constraint solver in a CLP language itself is that solver variables will usually be viewed quite differently inside the solver module than outside. For example, externally a solver variable might be seen as an object of type `cint` while internally the solver variable might be a fixed integer index into a global variable tableau, and hence have type `int`. Abstract exportation of the type ensures that the different views of the type are handled correctly, but there is also the issue of the different views of instantiations.

Solver variables with abstract types can only be seen externally as having instantiation states `new`, `old`, or `ground`. However, the internal view associated to instantiations `old` and `ground` might be different to the external view. For example, if the solver variable is represented as an integer into a global variable tableau, what is externally viewed as `old` is internally viewed as `ground`. The same can happen for instantiation `ground`. Consider the following solver type:

```
:- export_abstract typedef cvar -> ( variable(int) ; fixed(float)).
```

which implements a float constraint variable using two functors: `fixed` which contains the value of the variable if it is known to be fixed, and `variable` which contains a pointer into a global variable tableau, otherwise. In this case, what is externally view as `ground` can be more accurately described by

```
:- instdef fixed -> fixed(ground).
```

HAL handles this by means of the `base_insts`($Type, InstOld, InstGround$) declaration which provides the internal view of the `old` ($InstOld$) and `ground` ($InstGround$) instantiations for $Type$. For example,

```
:- base_insts(cvar,ground,fixed).
```

declares that the base instantiations `old` and `ground` for the type `cvar` are internally treated as `ground` and `fixed`.

Another complication of writing a constraint solver in a CLP language itself, is that we typically wish to be able to automatically coerce elements of the base-type into the solver type. Consider the following part of a simple integer bounds propagation solver:

[3] It seems clear to us that this will need to change in the future.

```
:- module bounds.
:- export_abstract typedef cint = ... %% internal cint representation
:- export instance fd_solver(cint).
:- export pred init(cint::no) is det.
:- export pred cint::oo = cint::oo is semidet.
:- export func coerce(int::in) --> cint::no is det.
```

We would then like to be able to write integer constants as `cint` arguments of predicates and functions, for example as in `X + 3*Y >= Z + 2`. Thus, we need to instruct the compiler to perform automatic coercion between an `int` and a `cint`. This is achieved by the existence of a declaration in the class `int_solver` which tells the compiler to support automatic coercion for instances of the method `coerce(int) --> S <= int_solver(S)`. The compiler then wraps integer constants in calls to coerce, and type inference determines if these are coercions or simply the identity. Thus the constraint above (assuming `X`, `Y` and `Z` are `old`) is translated to

```
T1=coerce(3), *(T1,Y,T2), +(X,T2,T3), T4=coerce(2), +(Z,T4,T5), >=(T3,T4).
```

The above translation may appear very inefficient, since many new solver variables are introduced, and many constraints each of which will involve propagation to solve. This is not necessarily the case. The solver could be defined so that `cint`s are structured terms, which are built by `coerce`, `+` and `*`, and only the constraint relations build propagators. Thus, the `+` function would be defined as

```
:- export_abstract typedef cint -> (var(bvar) ; int(int)
                                   ; plus(cint,cint) ; times(cint,cint) ).
:- export func coerce(int::in) --> cint::no.
coerce(I) --> int(I).
:- export func cint::oo + cint::oo --> cint::no.
X + Y --> plus(X,Y).
```

Using this scheme, the goal above builds up a structure representing the terms of the constraint and then the equality predicate simplifies the structure and implements the appropriate propagation behaviour.[4]

When implementing constraint solvers or search strategies it is vital for efficiency to be able to destructively update a global data structure which might, for example, contain the current constraints in solved form.

To cater for this, HAL provides global variables. These are local to a module and cannot be accessed by name from outside the module, much like C's `static` variables. Global variables behave as references. They can never be directly passed as an argument to a predicate; they are always de-referenced at this point. They come in two flavours: backtracking and non-backtracking. Non-backtracking global variables must be ground. For example:

```
:- VarNo glob_var int = 0.
init(V) :- V = $VarNo, $VarNo := $VarNo + 1.
```

[4] It seems possible to build code to automatically extend a simple solver (e.g. `bvar`) to support this more efficient interface.

defines a backtracking global variable `VarNo` of type `int` initialised to value 0. This variable can be used, for example, to keep track of the number of variables encountered so far.

A major reason for designing HAL to be as pure as possible it that it simplifies the use of powerful compile-time optimizations such as unfolding, reordering and many low level optimizations. Typically a solver, though implemented using impure features, presents a "pure" interface to other modules that use it. For example, although the solver data structures are stored in a global variable, the solver will "behave" the same regardless of the order of external calls to primitive constraints. The purity declarations allow the user to control the inheritance of purity.

HAL also provides Constraint Handling Rules (CHRs) for defining new solvers. These 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 [8].

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.

As in most implementations, HAL CHRs [13] 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.

In HAL, CHR constraints must have a mode which does not change the instantiation states 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.

The following code puts all of these elements together. It gives part of a Boolean solver implemented in HAL using CHRs.[5]

[5] Somewhat simplified for ease of exposition.

```
:- module bool_chr.
:- instance bool_solver(boolchr).
:- export_abstract typedef boolchr -> wrap(int).
:- base_insts(boolchr,ground,ground).
:- VNum glob_var int = 0.
:- export trust pure pred init(boolchr::no) is det.
init(V) :- V = wrap($VNum), $VNum := $VNum + 1.
:- export func coerce(bool::in) --> boolv::no is det
coerce(true) --> X :- init(X), (t(X) -> true ; error).
coerce(false) --> X :- init(X), (f(X) -> true ; error).
:- chr_constraints t/1, f/1, and/3, notboth/2.
:- export pred t(boolchr::oo) is semidet.
:- export pred f(boolchr::oo) is semidet.
t(X), f(X) <=> fail.
:- export pred and(boolchr::oo,boolchr::oo,boolchr::oo) is semidet.
t(X) \ and(X,Y,Z) <=> Y = Z.
t(Y) \ and(X,Y,Z) <=> X = Z.
f(X) \ and(X,Y,Z) <=> f(Z).
f(Y) \ and(X,Y,Z) <=> f(Z).
f(Z) \ and(X,Y,Z) <=> notboth(X,Y).
t(Z) \ and(X,Y,Z) <=> t(X), t(Y).
:- pred notboth(boolchr::oo,boolchr::oo) is semidet.
t(X) \ notboth(X,Y) <=> f(Y).
t(Y) \ notboth(X,Y) <=> f(X).
f(X) \ notboth(X,Y) <=> true.
f(Y) \ notboth(X,Y) <=> true.
```

In this case `boolchr`s are simply variable indices[6] 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. It must be declared `trust pure` to hide the use of impure global variables. Coercion returns a new `boolchr` which is appropriately constrained. The `chr_constraint` declaration lists which predicates are implemented by CHRs. The remaining parts are CHRs. The `t` and `f` constraints constrain a Boolean variable to have a fixed value. The first rule 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. Note that equality is also implemented using CHRs.

9 Current System

The HAL compiler, system and libraries consists of some 60,000 lines of HAL code (which is also legitimate SICStus Prolog code). HAL programs may be compiled to either Prolog or Mercury. Mercury compiles to C and makes use of the information in declarations to produce efficient code. However, better debugging facilities in Prolog and the ability to handle code without type and

[6] HAL does not yet support CHRs on Herbrand types

mode declarations have made compilation to Prolog extremely useful in the initial development of the compiler.

Currently, we require full type, mode, determinism, I/O and purity declarations for exported predicates and functions. The HAL compiler performs type checking and inference. Partial type information may be expressed by using a '?' in place of an argument type or class constraint. The type checking algorithm is based on a constraint view of types and is described in [5]. The HAL compiler currently does not perform mode inference, but does perform mode checking [9]. The compiler performs determinism and purity inference and checking, using its generic analysis engine [19].

Compilation into Mercury required extending and modifying the Mercury language and its runtime system in several ways. Some of these extensions have now been incorporated into the Mercury release. The first extension was to provide an "any" instantiation, corresponding loosely to HAL's `old` instantiation. The second extension was to add purity declarations, as well as "trust me" declarations indicating to the Mercury compiler that it should just trust the declarations provided by the user (in our case the HAL compiler). The third extension was to provide support for global variables. The backtracking version needs to be trailed, while for the non-backtracking version the data needs to be stored in memory which will not be reclaimed on backtracking. Another extension was to provide run-time support for different equality operations. In order to support polymorphic operations properly, Mercury needs to know how to equate two objects of a (compile-time unknown) type. Mercury provides support for comparing two ground terms, but we needed to add similar support for equating two non-ground terms, as well as overriding the default ground comparison code to do the right thing for solver types.

Currently the HAL system provides four standard solvers: one for integers, two for floats and a Herbrand solver for term equations. The Herbrand solver is more closely built into the HAL implementation than the other two solvers, with support at the compiler level to leverage from the built-in term equation solving provided by Prolog and Mercury. One complicating issue has been that, as discussed earlier, Mercury only provides restricted forms of equality constraints. Since we wished to support full equality constraints, this required implementing a true unification based solver which interacted gracefully with the Mercury run-time system. This integration is described more fully in [3].

The integer solver is the same as that described in [10]. It is a bounds propagation solver which keeps linear constraints in a tableau form, and simplifies them during execution to improve further propagation. It was originally embedded in CLP($\mathcal{R}$)'s compiler and runtime system, yielding the language CLP($\mathcal{Z}$). It has since been interfaced to Mercury, and then to HAL via the Mercury interface. The first float solver is the solver from CLP($\mathcal{R}$), interfaced in the same way, while the second uses CPLEX [14] via the C foreign function interface.

10 Conclusion

We have introduced HAL, a language which extends existing CLP languages by providing semi-optional declarations, a well-defined solver interface, dynamic scheduling and global variables. These combine synergistically to give a language which is potentially more efficient than existing CLP languages, allows ready integration of foreign language procedures, is more robust because of compile-time checking, and, most importantly, allows flexible choice of constraint solvers which may either be fully or partially written in HAL. An initial empirical evaluation of HAL is very encouraging.

Despite several programmer years of effort, much still remains to be done on the HAL implementation. An important extension is to provide mutable data structures. Currently, only global variables are mutable but we would also like non-global mutable variables. One way is to provide references; another is to provide `unique` and `dead` declarations as is done in Mercury. We will explore both. We wish to support solver dependent compile-time analysis and specialization of solver calls. This is important since it will remove most of the runtime overhead of constructing arguments for constraints.

Apart for these specific aims, there are number of broader areas we are involved with. We are investigating how to extend the modelling capabilities of HAL, beyond the type class mechanism, to give simpler modelling of typical problems. We are exploring many further opportunities for global analyses and optimizations based on these analyses. We are in the process of building and integrating more solvers in the HAL system. Other important future research directions are debugging of constraint programs, and extended capabilities for programming search.

Acknowledgements

Many people have helped in the development of HAL. In particular they include Warwick Harvey, Christian Holzbaur, David Jeffery, Nick Nethercote, David Overton and Peter Schachte. We would also like to thank Zoltan Somogyi, Fergus Henderson and other members of the Mercury development team who have supported our (mis)use of Mercury.

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. M. García de la Banda, D. Jeffery, K. Marriott, P.J. Stuckey, N. Nethercote, and C. Holzbaur. Building constraint solvers with HAL. In P. Codognet, editor, *Logic Programming: Proceedings of the 17th International Conference*, LNCS, pages 90–104. Springer-Verlag, 2001.
3. B. Demoen, M. García de la Banda, W. Harvey, K. Marriott, and P.J. Stuckey. Herbrand constraint solving in HAL. In D. De Schreye, editor, *Logic Programming: Proceedings of the 16th International Conference*, pages 260–274. MIT Press, 1999.

4. B. Demoen, M. García de la Banda, W. Harvey, K. Marriott, and P.J. Stuckey. An overview of HAL. In J. Jaffar, editor, *Proceedings of the Fourth International Conference on Principles and Practices of Constraint Programming*, LNCS, pages 174–188. Springer-Verlag, October 1999.
5. B. Demoen, M. García de la Banda, and P.J. Stuckey. Type constraint solving for parametric and ad-hoc polymorphism. In *Procs. of the 22nd Australian Comp. Sci. Conf.*, pages 217–228, 1999.
6. D. Diaz and P. Codognet. A minimal extension of the WAM for clp(fd). In *Procs. of ICLP93*, pages 774–790, 1993.
7. A.J. Fernández and B.C. Ruiz Jiménez. Una semántica operacional para CProlog. In *Proceedings of II Jornadas de Informática*, pages 21–30, 1996.
8. T. Früwirth. Theory and practice of constraint handling rules. *Journal of Logic Programming*, 37:95–138, 1998.
9. M. García de la Banda, P.J. Stuckey, W. Harvey, and K. Marriott. Mode checking in HAL. In J. LLoyd et al., editor, *Proceedings of the First International Conference on Computational Logic*, LNCS 1861, pages 1270–1284. Springer-Verlag, July 2000.
10. W. Harvey and P.J. Stuckey. Constraint representation for propagation. In *Procs. of PPCP98*, pages 235–249, 1998.
11. M. Hermenegildo, F. Bueno, D. Cabeza, M. García de la Banda, P. López, and G. Puebla. The CIAO multi-dialect compiler and system. In *Parallelism and Implementation of Logic and Constraint Logic Programming*. Nova Science, 1999.
12. C. Holzbaur. Metastructures vs. attributed variables in the context of extensible unification. In *Procs. of the PLILP92*, pages 260–268, 1992.
13. C. Holzbaur, P.J. Stuckey, M. García de la Banda, and D. Jeffery. Optimizing compilation of constraint handling rules. In P. Codognet, editor, *Logic Programming: Proceedings of the 17th International Conference*, LNCS, pages 74–89. Springer-Verlag, 2001.
14. ILOG. CPLEX product page. http://www.ilog.com/products/cplex/.
15. J. Jaffar, S. Michaylov, P. Stuckey, and R. Yap. The CLP($\mathcal{R}$) language and system. *ACM Transactions on Programming Languages and Systems*, 4(3):339–395, 1992.
16. D. Jeffery, F. Henderson, and Z. Somogyi. Type classes in Mercury. Technical Report 98/13, University of Melbourne, Australia, 1998.
17. S. Kaes. Parametric overloading in polymorphic programming languages. In *ESOP'88 Programming Languages and Systems*, volume 300 of *LNCS*, pages 131–141, 1988.
18. K. Marriott and P.J. Stuckey. *Programming with Constraints: an Introduction*. MIT Press, 1998.
19. N. Nethercote. The analysis framework for HAL. Master's thesis, Dept. of Comp. Sci and Soft. Eng, University of Melbourne, 2002.
20. Z. Somogyi, F. Henderson, and T. Conway. The execution algorithm of Mercury: an efficient purely declarative logic programming language. *Journal of Logic Programming*, 29:17–64, 1996.
21. P. Wadler and S. Blott. How to make *ad-hoc* polymorphism less *ad-hoc*. In *Proc. 16th ACM POPL*, pages 60–76, 1989.