An Overview of HAL

Bart Demoen[1], Maria García de la Banda[2], Warwick Harvey[2], Kim Marriott[2], and Peter Stuckey[3]

[1] Dept. of Computer Science, K.U.Leuven, Belgium
[2] School of Computer Science & Software Engineering, Monash University, Australia
[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 a well-defined solver interface, mutable global variables for implementing a constraint store, and dynamic scheduling for combining, extending and writing new constraint solvers. Equally importantly, HAL supports 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.

1 Introduction

Constraint logic programming (CLP) languages are evolving to support more flexible experimentation with constraint solvers. First generation CLP languages, such as CLP($\mathcal{R}$) [9], 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)` [3], provided more support by viewing the solver as a "glass box" which the programmer could extend to provide 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 (see e.g. [14]). 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 [4] and attributed variables [8]. Unfortunately, support for developing solvers in these languages is still less than satisfactory for the reasons detailed below.

We describe a new CLP language, HAL, which has been explicitly designed to support experimentation with different constraint solvers and development of new solvers. Our specific design objectives were four-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 procedures (in particular, 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 and when calling foreign language procedures, additional compile-time checking can detect common programming errors and so improve program robustness.
– *Flexible choice of constraint solvers:* It should be easy to "plug and play" with different constraint solvers over the same domain. Furthermore, 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 even for user-defined constraint solvers.

The second feature is a well-defined interface for solvers. Solvers are modules which provide various fixed predicates and functions for initializing variables and adding constraints. Obviously, such an interface supports "plug and play" experimentation with different solvers.

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.

The fourth feature is a provision for "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 the constraint solver writer to efficiently implement a persistent constraint store.

A well-defined solver interface and global variables are, to the best of our knowledge, novel in the context of CLP. While declarations and dynamic scheduling are not new, incorporating them into a CLP language which also allows user-defined constraint solvers has proven challenging. In isolation, each feature is relatively well understood; it is their combination that is not. Major difficulties have been to provide (limited) automatic coercion between types, compile-time reordering of literals during mode-checking with appropriate automatic initialization of solver variables, and efficient, yet accurate mode and determinism checking in the presence of dynamic scheduling. Dynamic scheduling has also complicated the design of the solver interface since the choice and implementation of delay conditions is necessarily solver dependent. One interesting feature has been the need to provide an external and internal view of the type and mode of a solver variable.

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 [13]. 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 and determinism 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 and determinism 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.

We know of only one other language that integrates declarations into a CLP language: CIAO [7]. The design of CIAO has proceeded concurrently with that of HAL. HAL has been concerned with providing a language for experimentation with constraint solvers. In contrast, CIAO has focused on exploring the design and use of more flexible declarations for program analysis, debugging, validation, and optimization, and on supporting parallelism and concurrency. Constraint solving in CIAO is inherited from the underlying &-Prolog/SICStus Prolog implementation: solvers are written using attributed variables. Thus, CIAO does not provide an explicit solver interface, does not provide a dual view of constraint variables, requires the programmer to explicitly insert initialization and coercion predicate calls, and, if more than one (non-Herbrand) constraint solver is used, the programmer must explicitly call the appropriate solver.

2 The HAL Language

In this section we provide an overview of the HAL language. The basic HAL syntax follows the standard CLP syntax, with variables, rules and predicates defined as usual (see, e.g., [11] for an introduction to CLP). Our philosophy has been to design a language which is as pure as possible, without unduly compromising efficiency. Thus, HAL does not provide many of the non-pure but standard logic programming built-ins. For instance, it does not provide database predicates; instead global variables can be used to provide most of this functionality.

The module system in HAL is similar to the Mercury module system. A module is defined in a file, it `imports` the modules it uses and has `export` annotations on the declarations for the objects that it wishes to be visible to those importing the module. Selective importation is also possible.

The base language supports integer, float, string, atom and term data types. However, the support is limited to assignment, testing for equality, and construction and deconstruction of ground terms. More sophisticated constraint solving on these types is provided by importing a constraint solver for the type.

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)
:- import simplex.                                                     (L2)
:- export pred mortgage(cfloat,cfloat,cfloat,cfloat,cfloat).          (L3)
:-        mode mortgage(in,in,in,in,out) is semidet.                  (L4)
:-        mode mortgage(oo,oo,oo,oo,oo) is nondet.                    (L5)
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 of the file ($L1$) states that this is the definition of the module `mortgage`. Line ($L2$) imports a previously defined module called `simplex`. This provides a simplex-based linear arithmetic constraint solver for constrained floats, called `cfloats`. Line ($L3$) declares that this module exports the predicate `mortgage` which takes five `cfloats` as arguments. This is the *type* declaration for `mortgage`. Lines ($L4$) and ($L5$) 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 ($L5$) 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 more flexible mode allows arbitrary uses of the mortgage predicate, but will be less efficient to execute. Line ($L4$) also states that for this mode `mortgage` is `semidet`, meaning that it either fails[1] or succeeds with exactly one answer.[2] For the second mode ($L5$) the determinism is `nondet` meaning that the query may return 0 or more answers. The rest of the file contains the standard two rules defining `mortgage`.

[1] For example `mortgage(0.0,-1.0,0.0,0.0,B)` fails

[2] HAL currently does not perform determinism analysis and Mercury is unable to confirm this determinism declaration.

2.1 Type, Mode and Determinism Declarations

As we can see from the above example, one of the key features of HAL is that programmers may annotate predicate definitions with type, mode and determinism declarations (modelled on those of Mercury). 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.

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's checker.

Type Declarations: These specify the representation format of a variable or argument. Thus, for example, the type system distinguishes between constrained floats (`cfloat`) and the standard numerical float (`float`) since these have a different representation.

Types are specified using type definition statements. They are (polymorphic) regular tree type statements. For instance,

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

Equivalence types are also allowed. For example,

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

where the right-hand side must be a type.

Overloading of predicates is allowed, although the predicate definitions for different type signatures must be in different modules. Overloading is important since it allows the programmer to overload the standard arithmetic operators and relations (including equality) for different types, allowing a natural syntax in different constraint domains.

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

```
:- module complex.
:- import simplex.
:- 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, the programmer can use 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 `simplex` 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.

One of the hardest issues we have faced in type checking and inference is the need to handle automatic coercion between types. For instance, in the definition of `mortgage` we have used the constraint `T >= 1.0`. The constraint `>=` expects to take two `cfloat`s whereas `1.0` is of type `float`. The compiler overcomes this typing problem by automatically inserting a call to a coercion function (defined in `simplex`) to coerce `1.0` to a `cfloat`. Such automatic coercion is important because it allows constraints and expressions to be written in a natural fashion. Currently, the type system for HAL only supports simple coercion, namely, a type can be involved in at most one coercion relationship. Even handling this restricted kind of coercion is difficult and is one of the most complex parts of the type checking and inference mechanism.

Mode Declarations: A mode is associated with an argument of a predicate. It has the form $Inst_1 \to Inst_2$ where $Inst_1$ describes the input instantiation state of the argument and $Inst_2$ describes the output instantiation state. The basic instantiation states for a solver variable are `new`, `old` and `ground`. Variable X is `new` if it has not been seen by the constraint solver, `old` if it has, and it is `ground` if X is constrained to take a fixed value.

For data structures, such as a list of solver variables, more complex instantiation states (lying between `old` and `ground`) may be used to describe the state of the data structure. Instantiation state definitions look something like type definitions. An example is

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

which is read as 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; `multidet` at least one solution; `semidet` at most one solution; `det` exactly one solution; `failure` no solutions; and `erroneous` a runtime error.

2.2 Constraint Solvers

Constraint solvers are implemented as modules in HAL. Selective importation of modules supports "plug and play" experimentation with different solvers over the same domain. For instance, if we define two solvers `intsolv1` and `intsolv2` for finite domain integer constraints, then we can compile a HAL program using either solver by changing which solver module is imported.

A constraint solver is a module that defines some type for the constrained variable, and predicates to initialize and equate these variables. Typically, it will also define a coercion function and a function for returning the value of a ground variable.

Consider implementing a simple integer bounds propagation solver. The following declarations can be used:

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

The type `cint` is the type of bounds variables. It might be defined, for example, as an integer indexing into a HAL global variable tableau, as an integer representing a CPLEX variable number, or as a C pointer for a solver implemented in C. Usually, a constraint solver type is exported abstractly. As discussed earlier, this ensures other modules cannot modify `cint` variables without calling module `bounds`.

The `init` predicate is used to initialize a variable, its mode is `no` or `new -> old`. The compiler will automatically add appropriate calls to the initialization predicate in user code that makes use of the `bounds` solver.

The equality predicate is required for all constraint solvers. In the above code for `cint`s it is annotated as `export_only`. This indicates that it should be visible to importing modules but not within this module. This is useful because inside the module `bounds` we manipulate the internal (rather than the external) representation of `cint`s using equality. We can, however, access the equality predicate by using explicit module qualification.

A complete `bounds` module would also export declarations for (primitive constraint) predicates such as `>=`, `<=`, `!=` as well as functions such as `+`, `-` and `*`.

As we have seen previously, it is useful to allow solver-dependent type coercion. In this case, we would 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`. The `coerce` declaration does this, indicating that the coercion function is `coerce_int_cint`. Thus the constraint above (assuming `X`, `Y` and `Z` are `old`) is translated to

```
coerce_int_cint(3,T1), *(T1,Y,T2), +(X,T2,T3),
coerce_int_cint(2,T4), +(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_int_cint`, `+` 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 cint + cint --> cint.
:-       mode oo + oo --> 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.

Herbrand Solvers: Most HAL types are structured data types defined in terms of constructors where elements of these data types correspond to terms in traditional CLP languages. For example, our earlier type definition defined the (polymorphic) `list` type in terms of the constructors `[]` (nil) and "`.`" (cons). As indicated previously, the HAL base language only provides limited operations for dealing with such data structures. In essence, it provides an equality predicate that only supports modes for constructing a new term, deconstructing a bound term or comparing two bound terms. This corresponds to the operations that Mercury allows on its data structures.

If the programmer wishes to use more complex constraint solving for some type, then they must explicitly declare they want to use the Herbrand constraint solver for that type. Conceptually, the HAL run-time system provides a Herbrand constraint solver for each term type defined by the programmer. This solver provides full equality (unification without occurs check), supporting the use of Prolog-like variables for structured data types and allowing programming idioms like difference lists. In practice, the constraint solver is implemented by automatically generating code for each functor in the type definition, and replacing Herbrand constraints by calls to this code.

For example, if the programmer wishes to use difference lists then they need to include the declaration:

```
:- herbrand list/1.
```

However, we note that in most cases data structure manipulation, even if the data structures contain solver variables, does not require the use of a Herbrand constraint solver.

Inside and Outside View of the Solver: Solver modules are more complicated than other modules because they define solver variables which will be viewed in two different ways. For example, outside the solver module a solver variable might be seen as a `cint` with `old` instantiation state, which cannot be modified without calling the solver. Inside the module the solver variable might be a fixed integer index into a global variable tableau, and hence have type `int` and instantiation `ground`.

Abstract exportation of the type ensures that the different views of type are handled correctly, but there is also the issue of the different views of modes. In particular, the exported predicates need to be viewed as having instantiations which are `old` while internally they are something different. To handle this we allow a renaming instantiation declaration, for example

```
:- reinst_old cint_old = ground.
```

declares that the instantiation name `cint_old` is treated as `ground` within this module (the solver's internal view), but exported as `old`. Typically, where a dual view of modes is required, the declarations for exported predicates are similar to the following:

```
:- modedef cint_oo = (cint_old -> cint_old).
:- export pred cint >= cint.
:-        mode cint_oo >= cint_oo is semidet.
```

2.3 Dynamic Scheduling

HAL includes a form of "persistent" dynamic scheduling designed specifically to support constraint solving. HAL's 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, the delayed goals remain active and are reexecuted whenever the delay condition becomes true again. This is useful, for example, if the delay condition is "the lower bound has changed." However, the delayed goals may contain calls to the special predicate `kill`. When this is executed, all delayed goals in the immediate surrounding delay construct are killed; that is, they will never be executed again.

For example, the following delay construct implements bounds propagation for the constraint $X \leq Y$. 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. The functions $lb(V)$, $ub(V)$ and $val(V)$ respectively return the current lower bound, upper bound and value for V, while the predicates *upd_lb*, *upd_ub* and *upd_val* update these.

```
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
```

One important issue is the interaction between dynamic scheduling and mode and determinism analysis. While it is possible to analyze programs with arbitrary dynamic scheduling, currently this is complex, time consuming and sometimes inaccurate. Instead, by judiciously restricting the kind of goals which can be delayed, we guarantee that standard mode and determinism analysis techniques will still be correct in the presence of delayed goals.

The first restriction is that the mode for non-local variables in a delayed goal must be either `in` (i.e. `ground -> ground`) or `oo` (i.e. `old -> old`). This means that execution of the delayed goal, regardless of when it occurs, will not change the instantiation of the current variables of interest.[3] (This also relies on the instantiations `old` and ground being "downward closed.")

The second restriction is that the delayed goal must either be `det` or `semidet`. This ensures that if the current goal wakes up a delayed goal, the delayed goal's determinism cannot change that of the current goal. This is correct because `det` goals cannot wake up delayed goals, since they are only allowed to change the instantiation states for new variables, and a new variable cannot have previously occurred in a delay construct.

Combining Constraint Solvers: HAL is designed to make it easy to combine constraint solvers to form new hybrid constraint solvers. Imagine combining an existing propagation solver (variable type `cint` in module `bounds`) and an integer linear programming solver (type `ilpint` in module `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 cause the constraints to be sent to both underlying solvers. Communication between the solvers is managed by delayed goals created 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.
:- import 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).
:- trust mode init(no) is det.
init(p(XB,XC)) :- init(XB), init(XC),
           (lbc(XB) ==> XC >= lb(XB) || ubc(XB) ==> ub(XB) >= XC ||
           fixed(XB) ==> XC = val(XB), kill).
```

Solver Support: Delayed goals in HAL execute when a delay condition is found to be true. Delay conditions are defined by a constraint solver, and it is the role of that constraint solver to determine when a delay condition has become true. Delay conditions are defined as a type attached to the solver. By exporting types for delay constructs to the user, the solver writer can build a "glass box" solver which the user can extend to new constraints. Similarly, delay constructs allow the implementation of features such as invariants [12].

As an example, the bounds propagation solver might support the delay conditions `ubc(V)`, `lbc(V)` and `fixed(V)` which are true, respectively, when variable

[3] More generally we could allow any mode of the form i `->` i.

V has an upper or lower bound change, or becomes fixed. In order to support delayed goals, the `bounds` module would need to include the following declarations:

```
:- export typedef dcon -> (ubc(cint) | lbc(cint) | fixed(cint)).
:- export delay dcon.
:- export_abstract typedef delay_id = ... % internal representation
:- export pred get_id(delay_id).
:-        mode get_id(out) is det.
:- export pred delay(list(dcon),delay_id,list(pred)).
:-        mode delay(in,in,same(list(pred is semidet))) is semidet.
:- export pred kill(delay_id).
:-        mode kill(in) is det.
```

The delay conditions are simply defined as an exported type. The `delay` declaration allows them to appear in the left hand side of the "==>" construct.

The predicate `get_id` returns a new delay ID for a newly encountered delay construct. The predicate `delay` is then called with the list of delay conditions, the delay ID of the construct, and the list of actions (closures) corresponding to each condition. A `kill/0` predicate in the delay construct is replaced with a call to `kill/1` whose argument is the delay ID. It is the solver's responsibility to ensure that the action is called when the appropriate delay condition fires, and to remove killed actions.

Delay constructs are automatically translated by the compiler to use these predicates. For example, the delay construct in module `combined` translates to

```
get_id(D), delay([lbc(XB),ubc(XB),fixed(XB)],D,
           [XC >= lb(XB), XC <= ub(XB), (XC = val(XB), kill(D))]).
```

2.4 Global Variables

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 both statically scoped and dynamically scoped global variables. However, these variables are local to a module and cannot be accessed by name from outside the module, so the statically scoped version is more akin to 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.

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. However, global variables break this purity, and so restrict the applicability of such optimizations. Typically a solver, though implemented using impure features, presents a "pure" interface to other modules that use it. To this end, HAL supports purity declarations which control the inheritance of an impurity from a predicate to the predicates that call it. By declaring exported primitive constraint predicates as pure, the code using them may still be pure, and hence amenable to more optimization.

3 Current System

The HAL compiler, system and libraries consists of some 24,000 lines (comments and blank lines excluded) of HAL code (which is also legitimate SICStus Prolog code). HAL programs may be compiled to either SICStus Prolog or Mercury. In the longer term only compilation to Mercury will be supported. Mercury compiles to C and makes use of the information in declarations to produce efficient code. However, better debugging facilities in SICStus Prolog and the ability to handle code without type and mode declarations have made compilation to SICStus Prolog extremely useful in the initial development of the compiler.

Currently, we require full type, mode and determinism 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. The type checking algorithm is based on a constraint view of types and is described in [2]. The HAL compiler currently does not perform mode inference, but does perform mode checking. Currently, determinism declarations are not checked by the HAL compiler but simply passed through to the Mercury compiler. They are ignored when compiling to SICStus Prolog.

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 three standard solvers: one for integers, one for reals 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 SICStus 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 [1].

The integer solver is the same as that described in [5]. 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 real solver is the solver from $CLP(\mathcal{R})$, interfaced in the same way.

3.1 Evaluation

We now give some feel for the performance of HAL programs using the current HAL compiler when compiling to Mercury. This is not intended to provide a comprehensive comparison with other CLP languages, but rather to act as a "sanity check" confirming that the implementation has adequate efficiency.

For each of the three solvers—Herbrand, integer and real—we have selected four to five standard benchmarks and compared them to another CLP system. For the Herbrand benchmarks we compare with SICStus Prolog 3.7.1 (compact code), while for the integer and real benchmarks we compare with $CLP(\mathcal{Z})$ [5] and $CLP(\mathcal{R})$ v1.02 respectively. Since $CLP(\mathcal{Z})$ and $CLP(\mathcal{R})$ use exactly the same underlying solvers as HAL, these comparisons are solver independent. The results are shown in Table 1.

The Herbrand benchmarks are executed both with and without garbage collection. HAL's garbage collection is provided by the (Mercury) conservative garbage collector. The integer and real benchmarks are executed without garbage collection since $CLP(\mathcal{Z})$ and $CLP(\mathcal{R})$ do not provide garbage collection. All timings are the best over 20 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. Note that while the Herbrand benchmarks `deriv`, `hanoi` and `qsort` are effectively equivalent to Mercury programs, all other benchmarks require constraint solving.

The Herbrand benchmarks are four standard Prolog benchmarks: `deriv`, `serialize`, `hanoi` and `qsort`. The last two are shown in two forms: using ground lists and `append`, and using difference lists. For each benchmark we give two SICStus Prolog versions: the original benchmark "SICStus (Orig.)" and a modified version "SICStus (Mod.)" in which built-ins not present in HAL (such as cut) have been replaced by their HAL equivalents (such as if-then-else). As expected the HAL system is significantly faster than SICStus because of the use of declarations. Difference lists are not such a big win for HAL; this is due to the immaturity of the Herbrand solver and the advantages of compiling `append` with declarations.

The integer benchmarks include two standard benchmarks `crypta` and `eq10`, a forward checking and (the more usual) generalized forward checking version of `queens` and a Hamiltonian path program from [5]. Even though the communication overhead to the $\mathcal{Z}$ solver is presently greater in HAL than in $CLP(\mathcal{Z})$, the efficiency is comparable. The improvement in `queens_fc` arises because of repeated data structure manipulation within the search.

The real benchmarks are: `mg_extend`, an extended version of the mortgage program [10]; `matmul`, matrix multiplication used backwards to invert a matrix; `fib`, Fibonacci run backwards; and `circ`, a circuit design program from [6]. The $CLP(\mathcal{R})$ system puts considerable effort into compiling arithmetic constraints

Benchmark	Preds	Lits	no garbage collection SICStus (Orig.)	SICStus (Mod.)	HAL	garbage collection SICStus (Orig.)	SICStus (Mod.)	HAL
deriv	1	27	1430	1710	309	5220	5270	510
serialize	5	19	1700	1770	389	6260	6180	560
hanoi	2	9	2130	1680	199	15000	7710	640
hanoi_difflist	2	8	1060	410	139	5090	1110	290
qsort	3	10	10210	6470	629	15710	11310	1510
qsort_difflist	3	10	10210	6400	689	15160	11220	1450

(a) Herbrand benchmarks

Benchmark	Pred	Lits	CLP($\mathcal{Z}$)	HAL
crypta	7	28	2990	2919
eq10	5	23	2610	2579
queens_fc	8	27	10840	8089
queens_gfc	9	28	6720	7669
hamil	13	29	223040	254549

(b) Integer benchmarks

Benchmark	Preds	Lits	CLP($\mathcal{R}$)	HAL
mg_extend	7	44	4900	5619
matmul	7	45	11833	4679
fib	1	10	3317	6229
circ	18	112	4050	1929

(c) Real benchmarks

Table 1. Empirical evaluation of HAL

efficiently, as opposed to the simple HAL interface. Hence, for `fib` where most time is spent adding constraints which are simple to solve, HAL is significantly slower. For the other benchmarks HAL is comparable, and it is faster when there is significant amounts of term manipulation (as in `matmul` and `circ`).

The preliminary results are very encouraging and augur well for the efficiency of new solvers implemented (perhaps partially) in HAL. Although the solver interfaces currently add significant overhead, much of this is due to the immaturity of HAL. We are confident that most of this overhead can be removed by relatively straightforward optimizations such as cross-module inlining.

4 Conclusion

We have introduced HAL, a new 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. Currently type checking and inference is supported but not mode inference or determinism analysis. One important extension is to provide support for type classes. These are a natural mechanism for defining a solver's capabilities, since a solver is (essentially) a module defining a certain set of predicates and functions. Another 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. Finally, 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.

Acknowledgements

Many people have helped in the development of HAL. They include Peter Schachte, and Fergus Henderson, David Jeffery and other members of the Mercury development team. We also thank members of the CIAO team for helpful discussions.

References

1. B. Demoen, M. García de la Banda, W. Harvey, K. Marriott, and P.J. Stuckey. Herbrand constraint solving in HAL. In *Procs. of ICLP99*, to appear.
2. 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.
3. D. Diaz and P. Codognet. A minimal extension of the WAM for clp(fd). In *Procs. of ICLP93*, pages 774–790, 1993.
4. T. Früwirth. Theory and practice of constraint handling rules. *Journal of Logic Programming*, 37:95–138, 1998.
5. W. Harvey and P.J. Stuckey. Constraint representation for propagation. In *Procs. of PPCP98*, pages 235–249, 1998.
6. N.C. Heintze, S. Michaylov, and P.J. Stuckey. CLP($\mathcal{R}$) and some electrical engineering problems. *Journal of Automated Reasoning*, 9:231–260, 1992.
7. 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.
8. C. Holzbaur. Metastructures vs. attributed variables in the context of extensible unification. In *Procs. of the PLILP92*, pages 260–268, 1992.
9. 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.
10. A. Kelly, A. Macdonald, K. Marriott, P.J. Stuckey, and R.H.C. Yap. Effectiveness of optimizing compilation of CLP($\mathcal{R}$). In *Procs. of JICSLP92*, pages 37–51, 1996.
11. K. Marriott and P.J. Stuckey. *Programming with Constraints: an Introduction.* MIT Press, 1998.
12. L. Michel and P. Van Hentenryck. Localizer: A modeling language for local search. In *Procs. of the PPCP97*, pages 237–251, 1997.
13. 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.
14. M. Wallace, editor. *CP98 Workshop on Large Scale Combinatorial Optimization and Constraints*, 1998. `http://www.icparc.ic.ac.uk/~mgw/chic2_workshop.html`.