Compiling Ask Constraints

Gregory J. Duck[1], María García de la Banda[2], and Peter J. Stuckey[1]

[1] Department of Computer Science and Software Engineering
The University of Melbourne, Vic. 3010, Australia
{gjd,pjs}@cs.mu.oz.au
[2] School of Computer Science and Software Engineering
Monash University, Vic. 3800, Australia
mbanda@csse.monash.edu.au

Abstract. In this paper we investigate how to extend a generic constraint solver that provides not only *tell* constraints (by adding the constraint to the store) but also *ask* tests (by checking whether the constraint is entailed by the store), with general ask constraints. Ask constraints are important for implementing constraint implication, extensible solvers using dynamic scheduling and reification. While the ask-test must be implemented by the solver writer, the compiler can extend this to provide ask behaviour for complex combinations of constraints, including constraints from multiple solvers. We illustrate the use of this approach within the HAL system.

1 Introduction

A constraint c of constraint domain $\mathcal{D}$ expresses relationships among variables of $\mathcal{D}$. All constraint programming frameworks (such as $CLP(\mathcal{D})$ [4, 5]) use c as a *tell* constraint, allowing the programmer to add the relationship to the current constraint store C and check that the result is possible satisfiable, i.e., $\mathcal{D} \models \tilde{\exists}(C \wedge c)$. However, some frameworks (such as `cc(`$\mathcal{D}$`)` [6]) also use c as an *ask* constraint, allowing the programmer to detect constraints stores C for which the relationship already holds, i.e., $\mathcal{D} \models C \rightarrow c$.

Ask constraints are often used to control execution by associating them to some goal, which is to be executed if and when the associated ask constraints succeed (i.e., become entailed by the constraint store).

Example 1. The following `cc` [6] definition of the constraint `min(X,Y,Z)`

```
min(X,Y,Z) :- X >= Y | Z = Y.
min(X,Y,Z) :- Y >= X | Z = X.
```

where `Z` is the minimum of `X` and `Y` is read as follows: when the `min` constraint is executed, wait until one of the ask constraints to the left of the bar `|` holds and then execute the tell constraint to the right of the bar. In the case of the `cc` framework the implementation also encodes a commit: once one ask constraint holds the other will never be reconsidered. The code implements a form of implication whose logical reading is:

$$min(x, y, z) \Leftrightarrow (x \geq y \rightarrow z = y) \wedge (y \geq x \rightarrow z = x) \qquad \boxdot$$

Note that it is not enough for the above framework to be able to test whether a constraint c is entailed by the current constraint store (this one-off test will be referred to in the future as the *ask-test*). It also needs to detect changes in the constraint store that might affect the entailment of c, so that the ask-test can be re-executed. Hence, ask constraints are strongly connected to logical implication. In fact, it is this connection that makes them so useful for implementing many important language extensions, such as those involving constraint solvers.

In this paper we consider a language that supports an ask construct of the form $(F_1\texttt{==>}G_1\texttt{\&}\ \ldots\texttt{\&}\ F_n\texttt{==>}G_n)$, where each F_i is a complex formula over constraints. The construct waits until some F_i is entailed by the store, and then executes its associated goal G_i. Several other languages, such as SICStus and ECLiPSe, implement related constructs for dynamic scheduling. However, they are typically hard-coded for a single solver, a pre-defined set of test conditions and do not support handling of (explicit) existential variables. Also, they usually only support formulas F made up of a single ask test condition. These restrictions considerably simplify the implementation of the construct.

This paper discusses the compilation of an ask construct with arbitrary ask-constraints, that allows the programmer to write code which closely resembles the logical specification. In particular, our contributions are as follows:

– We show how to extend an ask-test implemented by some underlying solver to a full *ask constraint* supporting dynamic scheduling.
– We show how to compile complex ask constraints which include existential variables and involve more than one solver, to the primitive ask-tests supported by the solvers.
– We show that the approach is feasible using an implementation in HAL [1].

2 Ask Constraints as High-level Dynamic Scheduling

This section formalizes the syntax, logical semantics and operational semantics of our ask construct. Its basic syntax is as follows:

```
( <ask-formula>_1 ==> goal_1 &  ... &  <ask-formula>_n ==> goal_n )
```

where an <*ask-formula*>$_i$ is a formula made up of primitive ask constraints, disjunction, conjunction, and existential quantification. Formally:

```
<ask-formula> := <ask-constraint>                      (primitive constraint)
<ask-formula> := <ask-formula> ';' <ask-formula>       (disjunction)
<ask-formula> := <ask-formula> ',' <ask-formula>       (conjunction)
<ask-formula> := exists <var-list> <ask-formula>       (existential quantification)
```

where <*ask-constraint*> represents some primitive ask constraint provided by a solver, <*var-list*> is a list of variables, and each *ask-constraint*/*goal* pair is referred to as an *ask branch*.

The operational semantics of the above construct is as follows. As soon as the solver(s) determine that an ask-formula is entailed, the corresponding delayed goal is called. The remaining ask branches will then never be considered, thus effectively *committing* to one branch. This commit behaviour can be easily avoided by specifying separate ask constructs for each branch. Note that ask constraints are *monotonic*, i.e., once they hold at a point during a derivation, they will always hold for the rest of the derivation. The advantage of monotonicity is that delayed goals need only be executed once, as soon as the associated ask constraints are entailed.

The declarative semantics of an ask branch F`==>`G is simply logical implication $F \rightarrow G$. The semantics of the whole construct (F_1`==>`G_1`&` $\cdots$`&` F_n`==>`G_n) is a conjunction of implications, *but* in order to agree with the commit the programmer must promise that the individual implications agree. That is, that for program P:

$$\mathcal{D} \wedge P \models (F_i \wedge F_j) \rightarrow (G_i \leftrightarrow G_j)$$

In other words, if the ask construct wakes on the formula F_i causing G_i to execute, and later formula F_j is implied by the store, then G_j is already entailed by G_i and need not be executed. Note that under these conditions the commit is purely used for efficiency, it will not change the logical semantics of the program, although it may of course change the operational behaviour since the underlying solvers are likely to be incomplete.

Example 2. Consider the following implementation of the predicate `either(X,Y)` which holds iff `X` or `Y` are true:

```
either(X,Y) :- ( X = 0 ==> Y = 1   &    Y = 0 ==> X = 1).
```

The logical reading is $(X = 0 \rightarrow Y = 1) \wedge (Y = 0 \rightarrow X = 1)$ which is equivalent to $(X = 1 \vee Y = 1)$ in the Boolean domain. If both ask constraints are made true $(X = 0 \wedge Y = 0)$, the right hand sides are equivalent $(Y = 1 \leftrightarrow X = 1)$ to false. ⊡

The proposed ask construct is indeed quite versatile. The following examples show how to use it to implement reification constraints, build constraints that involve more than one solver, and implement negation.

Example 3. A reified constraint $b \Leftrightarrow c$ constrains the Boolean variable b to be true if c is implied by the store, and b to be false if $\neg c$ is implied by the store, and vice versa. Consider defining a predicate $B \leftrightarrow \exists Y.X = [Y, Y]$ which "reifies" the right hand side. Note that the right hand side is equivalent to $\exists E1 \exists E2.X = [E1, E2], E1 = E2$. This can be implemented using ask constraints as

```
reifcomp(B,X) :-
    ( B=0, (exists [E1,E2] X = [E1,E2]) ==> X=[E1,E2], E1≠E2
    & B=1 ==> X=[Y,Y]
    & exists [Y] X=[Y,Y] ==> B=1
    & X=[] ; (exists [E1] X=[E1]) ;
             (exists [E1,E2,R] X=[E1,E2|R], (E1≠E2 ; R≠[])) ==> B=0)
```

These definitions assume X only takes on list values. ⊡

Example 4. The following program defines a length constraint which involves variables from a finite domain constraint solver, and from a Herbrand constraint solver for lists, and propagates information from one to the other:

```
length(L,N) :- ( N = 0 ; L = [] ==> N = 0, L = []
               & N >= 1 ; (exists [U1,U2] L = [U1|U2]) ==>
                          L = [_|L1], N >= 1, length(L1,N-1)).
```

⊡

Example 5. Consider the following definition of disequality

```
neq(X,Y) :- (X = Y ==> fail).
```

This (very weak) implementation of disequality waits until the arguments are constrained to be equal and then causes failure. ⊡

3 Compiling Primitive Ask Constructs

Let us now examine how to compile a primitive ask construct (i.e., one in which the left hand side of every ask branch is a single ask constraint) to the low-level dynamic scheduling supported by HAL.[3]

3.1 Low-level Dynamic Scheduling in HAL

HAL [1] provides four low-level type class methods that can be combined to implement dynamic scheduling: `get_id(Id)` which returns an unused identifier for the delay construct; `delay(`$event_i$`,Id,`$goal_1$`)` which takes a solver event, an id and a goal, and stores the information in order to execute the goal whenever the solver event occurs; `kill(Id)` which causes all goals delayed for the input id to no longer wake up, and `alive(Id)` which succeeds if the input id is still alive. In order for a constraint solver in HAL to support delay, it must provide an implementation of the `delay/3` method. Implementations of `get_id/1`, `kill/1` and `alive/1` can either be given by the solver or be re-used from some other source (such as the built-in system implementations). For more details see e.g. [1].

There are three major differences between HAL's dynamic scheduling and our ask construct. First, solver events ($event_i$) are single predicates representing an event in the underlying solver, such as "the lower bound of variable X changes". No conjunction, disjunction or existential quantification of events is allowed. Second, solver events need not be monotonic. Indeed, the example event "lower bound has changed" is clearly not. And third, a delayed goal will be re-executed *every time* its associated solver event occurs, until its id is explicitly killed.

Example 6. A finite domain integer (`fdint`) solver in HAL supporting dynamic scheduling typically provides the following solver events:

`fixed(V)`	The domain of V reduces to a single value;
`lbc(V)`	The lower bound of V changes (increases);
`ubc(V)`	The upper bound of V changes (decreases);
`dc(V)`	The domain of V changes (reduces).

[3] The compilation scheme can be adapted straightforwardly to other dynamic scheduling systems supporting delay identifiers.

Note that solver events do not need to be mutually exclusive: if the domain $\{1, 3, 5\}$ of X changes to $\{1\}$, the events `fixed(X)`, `ubc(X)` and `dc(X)` all occur.

Using the above events, a bounds propagator for the constraint $X \geq Y$ can be written as

```
geq(X,Y) :- get_id(Id),delay(lbc(Y),Id,set_lb(X,max(lb(Y),lb(X)))),
            delay(ubc(X),Id,set_ub(Y,min(ub(X),ub(Y)))).
```

where `lb` (and `ub`) are functions returning the current lower (and upper) bound of their argument solver variable. Likewise, `set_lb` (and `set_ub`) set the lower (and upper) bound of their first argument solver variable to the second argument. The code gets a new delay id, and creates two delaying goals attached to this id. The first executes every time the lower bound of Y changes, enforcing X to be greater than this bound. The second implements the reverse direction. ⊡

In addition to the four methods introduced above, HAL supports an "`asks`" declaration initially introduced into HAL to support the compilation of constraint handling rules that interact with arbitrary solvers [2]. The declaration allows constraints solvers to declare the relationship between a predicate implementing constraint c as a tell constraint, the predicate implementing c as an ask-test (a one-off test), and the list of solver events which might indicate the answer to the ask-test has changed and, therefore, the ask-test should be re-executed. Its syntax is as follows:

`:-` <*ask-test*> `asks` <*tell-constraint*> `wakes` <*wakes-list*>`.`

Example 7. The finite domain solver introduced in Example 6 might define the ask-test and declaration for the `geq` constraint as follows.

```
:- ask_geq(X,Y) asks geq(X,Y) wakes [lbc(X),ubc(Y)].
ask_geq(X,Y) :- lb(X) >= ub(Y).
```

The predicate `ask_geq` defines the ask-test for the `geq` constraint, and should be revisited when either the lower bound of X or the upper bound of Y change. ⊡

3.2 From Ask-Tests to Primitive Ask Constructs

The low-level dynamic scheduling of HAL allows us to compile the primitive ask construct:

$$(c_1(A^1_1, ..., A^1_{m_1}) \texttt{ ==> } Goal_1 \texttt{ \& } \ldots \texttt{ \& } c_n(A^n_1, ..., A^n_{m_n}) \texttt{ ==> } Goal_n)$$

if for each $c_i, 1 \leq i \leq n$, there exists the associated asks declaration:

`:-` $ask\_c_i(X_1, ..., X_{m_i})$ `asks` $c_i(X_1, ..., X_{m_i})$ `wakes [`$event_{i1}, ..., event_{in}$`].`

This is done by replacing the ask construct with:

`get_id(Id),delay_c`$_1$`(`$A^1_1, ..., A^1_{m_1}$`,Id,`$Goal_1$`),...,delay_c`$_2$`(`$A^n_1, ..., A^n_{m_n}$`,Id,`$Goal_n$`)`

and generating for each `delay_c`$_i$ the following code:

```
delay_c_i(X_1,...,X_m_i,Id,Goal) :-
   ( alive(Id) -> ( ask_c_i(X_1,...,X_m_i) -> kill(Id), call(Goal)
                  ; Retest = retest_c_i(X_1,...,X_m_i,Id,Goal)
                    delay(event_i1,Id,Retest), ... delay(event_in_i,Id,Retest))
   ;              true ).
retest_c_i(X_1,...,X_m_i,Id,Goal) :-
   ( ask_c_i(X_1,...,X_m_i) -> kill(Id), call(Goal) ; true).
```

The code for `delay_c`$_i$ first checks if the `Id` is alive, and if so determines whether or not the constraint already holds by calling the ask-test `ask_c`$_i/n$. If so, the `Id` is killed, the `Goal` is immediately called, and no other action is necessary. If the constraint is not yet entailed, a closure for the retesting predicate `retest_c`$_i$ is associated to each of the relevant solver events so that, each time a relevant solver event occurs, `retest_c`$_i$ is executed. This predicates checks whether the ask-test now succeeds and, if so, kills the `Id` and executes the goal. Note that the `delay` predicate for each solver used in the ask construct must support the same delay id type.

Example 8. Consider the asks declaration of Example 7. The compilation of

```
min(X,Y) :- ( geq(X,Y) ==> Z = Y & geq(Y,X) ==> Z = X).
```

results in

```
min(X,Y) :- get_id(Id), delay_geq(X,Y,Id,Z = Y), delay_geq(Y,X,Id,Z = X).
delay_geq(X,Y,Id,Goal) :-
  (alive(Id) -> ( ask_geq(X,Y) -> kill(Id), call(Goal)
                ; Retest = retest_geq(X,Y,Id,Goal),
                  delay(lbc(X),Id,Retest),delay(ubc(Y),Id,Retest))
  ;             true ).
retest_geq(X,Y,Id,Goal):- ( ask_geq(X,Y) -> kill(Id), call(Goal) ; true).
```
□

4 Compiling Disjunctions and Conjunctions

Conjunctions and disjunctions in ask formulae can be compiled away by taking advantage of the following logical identities:

1. Disjunctive implication: $(a \vee b) \to c$ is equivalent to $(a \to c) \wedge (b \to c)$; and
2. Conjunctive implication: $(a \wedge b) \to c$ is equivalent to $(a \to (b \to c))$.

Disjunctive implication is used to replace the branch

```
<ask-formula>_1 ; <ask-formula>_2 ==> Goal
```

in a construct, by the two branches

```
<ask-formula>_1 ==> Goal & <ask-formula>_2 ==> Goal
```

The two programs are operationally equivalent: the delayed `Goal` will be called once, after either <*ask-formula*>$_1$ or <*ask-formula*>$_2$ (whichever is first) hold. Similarly, conjunctive implication is used to replace the construct

```
<ask-formula>_1 , <ask-formula>_2 ==> Goal
```

by the construct:

```
( <ask-formula>1 ==> ( <ask-formula>2 ==> Goal ) )
```

Again, the new code is operationally equivalent: the delayed goal `Goal` will only be called once, after both <*ask-formula*>$_1$ and <*ask-formula*>$_2$ hold.

Note that the above simple conjunctive transformation cannot be directly applied to a branch appearing in a construct with 2 or more branches, because of the interaction with commit (the entire construct would be killed as soon as <*ask-formula*>$_1$ held, even if <*ask-formula*>$_2$ never did). We can solve this problem by using a newly created (local) delay id (`LId`) representing the delay on <*ask-formula*>$_1$ while using the original (global) delay id (`GId`) for the internal delay (since if <*ask-formula*>$_2$ also holds, the whole ask construct can commit).

An added complexity is that, for efficiency, we should kill the local delay id `LId` whenever `GId` is killed (if, say, another branch commits) so that the low-level HAL machinery does not re-execute (<*ask-formula*>$_2$ `==> Goal`) every time the events associated to <*ask-formula*>$_1$ become true. In order to do so we introduce the predicate `register(LId, GId)` which links `LId` to `GId`, so that if `GId` is ever killed, `LId` is also killed.

Example 9. Consider the compilation of

```
p(X,Y,Z,T) :- ( (X >= Y ; X >= Z) ==> Z = T & (Y >= X, Z >= X) ==> X = T).
```

The resulting code is

```
p(X,Y,Z,T) :- get_id(GId),
              delay_geq(X,Y,GId,Z = T), delay_geq(X,Z,GId,Z = T),
              get_Id(LId), register(LId,GId),
              delay_geq(Y,X,LId,delay_geq(Z,X,GId,X = T)).
```
⊡

By iteratively applying these rules, we can remove all conjunctions and disjunctions from ask formulae (without existential quantifiers).

5 Normalization and Existential Quantification

One of the first steps performed by HAL during compilation is program normalization, which ensures that every function and predicate has variables as arguments. The normalization exhaustively applies the following rules:

1. Rewrite $\exists \bar{x}.C \wedge y = f(t_1, \ldots, t_i, \ldots, t_n)$ where f is an n-ary function and t_i is either a non-variable term or a variable equal to some other $t_j, j \neq i$, to $\exists \bar{x} \exists v.C \wedge v = t_i \wedge y = f(t_1, \ldots, v, \ldots, t_n)$, where v is a new variable.
2. Rewrite $\exists \bar{x}.C \wedge c(t_1, \ldots, t_i, \ldots, t_n)$ where c is an n-ary constraint symbol and t_i is either a non-variable term or a variable equal to some other $t_j, j \neq i$, to $\exists \bar{x} \exists v.C \wedge v = t_i \wedge c(t_1, \ldots, v, \ldots, t_n)$ where v is a new variable.

Example 10. Consider the following definition of a *before-or-after* constraint for two tasks with start times `T1` and `T2` and durations `D1` and `D2` implements $T2 \geq T1 + D1 \vee T1 \geq T2 + D2$ without creating a choice.

```
before_after(T1,D1,T2,D2) :- (T1 + D1 > T2 ==> T1 >= T2 + D2),
                             (T2 + D2 > T1 ==> T2 >= T1 + D1).
```

which will have the body normalized into:

```
(exists [U1] U1 = +(T1,D1), U1 > T2 ==> U2 = +(T2,D2), T1 >= U2),
(exists [U3] U3 = +(T2,D2), U3 > T1 ==> U4 = +(T1,D1), T2 >= U4).
```

thus adding the existentially quantified variables U1, ..., U4. While the explicit quantification can be omitted for the variables appearing in the *tell* constraints on the right hand side (U2 and U4), this is *not* true for the *ask* constraints, since the (implicit) existential quantifier escapes the negated context of the ask. ⊡

Unfortunately, it is in general impossible to compile existential formulae down to primitive ask constraints. Only the solver can answer general questions about existential formulae.

Example 11. Consider an integer solver which supports the constraint `X > Y` and the function `X = abs(Y)` (which constrains X to be the absolute value of Y). The following ask construct `( exists [N] abs(N) = 2, N > 1 ==>` *Goal* `)` will always hold. However, it is impossible to separate the two primitive constraints occurring in the ask formula. Instead, we would have to ask the solver to treat the entire conjunction at once. ⊡

Thankfully, although normalization can lead to proliferation of existential variables in ask formulae, in many cases such existential variables can be compiled away without requiring extra help from the solver. Consider the expression

$$(\exists \bar{x} \exists v. v = f(y_1, \ldots, y_n) \land C) \rightarrow G$$

If f is a total function, such a v always exists and is unique. Thus, as long as none of the variables $y_1, \ldots, y_n$ are existentially quantified (i.e appear in $\bar{x}$) we can replace the above expression by the equivalent one

$$\exists v. v = f(y_1, \ldots, y_n) \land ((\exists \bar{x}. C) \rightarrow G)$$

Example 12. Returning to Example 10, we can transform the body code to

```
U1 = +(T1,D1), (U1 > T2 ==> U2 = +(T2,D2), T1 >= U2),
U3 = +(T2,D2), (U3 > T1 ==> U4 = +(T1,D1), T2 >= U4).
```

which does not require existential quantifiers in the ask-formula. ⊡

There are other common cases that allow us to compile away existential variables, but require some support from the solver. Consider the expression

$$(\exists \bar{x} \exists v. v = f(y_1, \ldots, y_n) \land C) \rightarrow G$$

where f is a partial function and none of the variables $y_1, \ldots, y_n$ appears in $\bar{x}$. This is equivalent to

$$(\exists v. v = f(y_1, \ldots, y_n)) \rightarrow (\exists v. v = f(y_1, \ldots, y_n) \land (\exists \bar{x}. C \rightarrow G))$$

The result follows since if there exists a v of the form $f(y_1, \ldots, y_n)$, then it is unique. Hence, the function f in the context of this test is effectively total. This may not seem to simplify compilation, but if we provide an ask version of the constraint $\exists v, v = f(y_1, \ldots, y_n)$ then we can indeed simplify the resulting code.

Example 13. Assuming we are dealing with integers, the expression $(x + 2^y \geq 2 \rightarrow b = 1)$ is equivalent to $(\exists z'.z' = 2^y) \rightarrow (\exists z.z = 2^y \wedge (x + z \geq 2 \rightarrow b = 1))$. If the compiler knows that the constraint $\exists z'.z' = 2^y$ is equivalent to $y \geq 0$, compilation of the code

```
g(X,Y,B) :- (X + 2^Y ≥ 2 ==> B = 1)
```

results in `g(X,Y,B) :- (Y ≥ 0 ==> (Z = 2^Y, (X + Z ≥ 2 ==> B = 1)).` ⊡

To use this simplification we need versions of the ask constraint for partial functions. These can be provided using the already introduced mechanisms for mapping tell constraints to ask constraints. For example, the mapping for $z = 2^y$ for a finite domain solver can be defined as

```
:- nonneg(Y) asks exists [Z] Z = 2^Y wakes [lbc(Y)].
nonneg(Y) :- lb(Y) >= 0.
```

To apply either of the simplifications above we also require information about total and partial functions. The HAL compiler already receives this information from the solver in terms of mode declarations. Example mode declarations that show the totality of `+` and the partialness of `^` are:

```
:- mode in + in ---> out is det.
:- mode in ^ in ---> out is semidet.
```

Partial functions are common in Herbrand constraints. Consider the constraint $x = f(y_1, \ldots, y_n)$, where f is a Herbrand constructor. This constraint defines, among others, a partial (deconstruct) function f_i^{-1} from x to each $y_i, 1 \leq i \leq n$. For this reason the compiler produces new ask tests `bound_f(X)` for each Herbrand constructor f, which check whether X is bound to the function f. Herbrand term deconstructions are then compiled as if the asks declaration

```
:- bound_f(X) asks exists [Y1,..,Yn] X = f(Y1,..,Yn) wakes [bound(X)]
```

appeared in the program. Note that in order to use this form of the ask constraint we may have to introduce further existential variables.

Example 14. Consider the compilation of the fragment `(exists [Y] X = [Y|Z] ==> p(X,Z))`. Although neither transformation seems directly applicable we can replace $\exists Y.X = [Y|Z]$ by the equivalent $\exists Y \exists V.X = [Y|V] \wedge V = Z$ and then use the partial function compilation to obtain

```
'bound_[|]'(X) ==> (X = [Y|V], (V = Z ==> p(X,Z)))
```

⊡

6 Compiling Equality

The general compilation scheme presented in previous sections assumes the existence of a simple mapping between an ask-test, and a set of solver events which indicate the answer to the ask-test may have changed. However, this is not always true, specially when dealing with structures that mix variables of different solvers. Consider testing the equality of lists of finite domain integers. To do so requires consulting both the Herbrand `list` solver and the `fdint` solver.

This problem is exacerbated by the fact that HAL supports polymorphic types, where some part of the type may only be known at run-time. Currently, the only solvers in HAL which include other solver types are Herbrand solvers and, thus, we will focus on them. However, the same issues arise if we wish to build sequence, multiset or set solvers over parametric types.

The problem of multiple types already arises when defining the ask-test version of equality (`==/2`). We can solve this problem in HAL by using type classes, i.e., by defining a type class for the `==/2` method and letting the compiler generate instances for this method for each Herbrand type.

Example 15. The code generated by the HAL compiler for the implementation of method `==/2` in the case of the `list(T)` type is as follows:

```
X == Y :- ( (var(X) ; var(Y)) -> X === Y
          ; X = [], Y = []    -> true
          ; X = [X1|X2], Y = [Y1|Y2], X1 == Y1, X2 == Y2 ).
```

where `===/2` succeeds if its arguments are (unbound) identical variables. ⊡

Extending the above ask-test to an ask constraint has two problems. First the only solver events currently supported by a Herbrand solver are: `bound(X)`, which occurs if X is bound to a non-variable term; and `touched(X)`, which occurs if the variable X is bound or unified with another variable (which also has events of interest). The reason why these are the only events supported is because they are the only ones that are independent of the type of the subterms of X. Thus, the asks declaration can only be defined as:

```
:- X == Y asks X = Y wakes [touched(X),touched(Y)]
```

which results in a very weak behaviour since it only notices changes at the topmost level of X and Y. For example, the goal `?- neq(X,Y), X = f(U), Y = f(V), U = V.` will not fail since even though X and Y are in the end identical, the unification of U and V does not create a solver event to retest the equality.

It is possible, though complex, to use overloading to introduce a new overloaded solver event `changed(X)` which occurs if any subterm of X is changed in some way (including unification with a variable). We could then provide an asks declaration

```
:- X == Y asks X = Y wakes [changed(X),changed(Y)]
```

which does not suffer from the above problem. However, there is a second problem which affects both solutions: repeatedly calling `==/2` from the top of the term is inefficient for large terms. A more efficient solution is to only partially retest.

For this we introduce a new ask-test for Herbrand terms, and show how it can be used to implement an efficient ask-test version of `==/2`. The ask-test is `samefunctor(X,Y)` which holds if X and Y have the same top-level functor, and can be implemented in Prolog as follows:

```
samefunctor(X,Y) :- (var(X), X == Y -> true ;
                     nonvar(X), nonvar(Y), functor(X,F,A), functor(Y,F,A)).
```

The advantage of this simple ask-test is that it only needs to be re-tested when `touched(X)` or `bound(X)`, i.e., its asks declaration can be defined as :

```
:- samefunctor(X,Y) asks samefunctor(X,Y) wakes [touched(X),bound(Y)].
```

indicating that we need to recheck the ask-test if X is bound to another variable (which fires the `touched(X)` event), since it could have been bound to Y,[4] and if X or Y is bound (the `touched(X)` event will fire if X is bound). Note that the ask-test has no corresponding tell version.

Let us now see how an efficient ask-test version for `==/2` can be written using `samefunctor`. Suppose the type of the Herbrand terms is $g(t_1, \ldots, t_n)$ where $t_1, \ldots t_n$ are types, and the type constructor g/n has $f_1/m_1, ..., f_k/m_k$ functor/arities. Then, the following implements (a high level view of) predicate `'delay_==_`g`_`n`'` which waits until two terms `X` and `Y` of type $g(t_1, \ldots, t_n)$ are equal to execute `Goal`. `(Id @ AskConstruct)` indicates the `Id` that should be given to the `delay` predicate resulting from each branch in `AskConstruct`.

```
'delay_==_g_n'(X,Y,GId,Goal) :-
   (alive(GId) -> get_id(LId), register(LId,GId),
      LId @ ( samefunctor(X,Y) ==>
         ( var(X) -> kill(GId), call(Goal)
         ; X = f1(X1,...,Xm1), Y = f1(Y1,...,Ym1),
            GId @ ( X1 = Y1, ..., Xm1 = Ym1 ==> kill(GId), call(Goal))
         ; ...
         ; X = fk(X1,...,Xmk), Y = fk(Y1,...,Ymk),
            GId @ ( X1 = Y1, ..., Xmk = Ymk ==> kill(GId), call(Goal))
   )) ; true ).
```

The code works as follows. If `GId` is alive, first a new local delay id `LId` is created for delay on `samefunctor`, and this is registered with `GId`. The whole body delays on the `samefunctor` ask constraint. When that holds, we test whether the variables are identical (true if either is a variable) and, if so, fire the goal. Otherwise, the two functors must be the same. Thus, we find the appropriate case and then delay on the conjunction of equality of the arguments. Here we can use the global delay identifier `GId` as the delay id for the ask formulae appearing for the arguments since at most one will be set up. The compilation of these conjunctions will, of course, introduce new local identifiers. When and if the arguments become equal, `Goal` will be called. Note that if the constructor $\mathtt{f}_i/m_i$ has arity zero (i.e. $m_i = 0$), then there are no arguments to delay until equal, and the goal will be immediately called.

The outermost ask construct code contains no explicit delay on equality, hence it can be compiled as described in the previous sections. The inner ask constructs do contain equality, and will be recursively handled in the same way.

Example 16. The generated code for the type `list(T)` is:

[4] We do not need to delay on `touched(Y)`, since if `touched(Y)` occurs, causing X and Y to become identical variables then `touched(X)` must have also occurred.

```
'delay_==_list_1'(X,Y,GId,Goal) :-
  (alive(Id) -> get_id(LId), register(LId,GId),
                delay_samefunctor(X,Y,LId,'delay_==_list_1_b'(X,Y,GId,Goal)
  ;             true).
'delay_==_list_1_b'(X,Y,GId,Goal) :-
  ( var(X) -> kill(GId), call(Goal)
  ;           X = [], Y = [], kill(GId), call(Goal)
  ;           X = [X1|X2], Y = [Y1|Y2], get_id(LId), register(LId,GId),
              'delay_=='(X1,X2,LId,'delay_=='(X2,Y2,GId,Goal))    ).
```
⊡

In order for the solution to work for polymorphic types, the `'delay_=='` predicate is defined as a method for a corresponding type class. HAL automatically generates the predicate `'delay_==_`g`_`n`'` for every Herbrand type constructor g/n that supports delay and creates an appropriate instance. For non-Herbrand solver types, the instance must be created by the solver writer. Normal overloading resolution ensures that at runtime the appropriate method is called.

This solution kills two birds with one stone. Firstly, it resolves the problems of delaying on equality by generating specialized predicates for each type. Secondly, because the predicate `'delay_=='` is overloaded, delay on equality is now polymorphic. Thus, it is possible to implement a truly polymorphic version of, for example, the `neq`/2 constraint. We can similarly implement a polymorphic ask constraint for disequality.

7 Experimental Results

The purpose of our experimental evaluation is to show that compiling ask constraints is practical, and to compare performance with hand-implemented dynamic scheduling where applicable. In order to do so, a simple prototype ask constraint compiler has been built into HAL. It does not yet handle existential quantifiers automatically. In the future we plan to extend the compiler to do this and also optimize the compilation where possible. All timings are the average over 10 runs on a Dual Pentium II 400MHz with 648M of RAM running under Linux RedHat 9 with kernel version 2.4.20 and are given in milliseconds.

The first experiment compares three versions of a Boolean solver written by extending a Herbrand constraint solver. The first, *hand*, is implemented using low-level dynamic scheduling (no compilation required). This is included as the ideal "target" for high-level compiled versions. The second, *equals*, implements the Boolean solver by delaying on equality, much like the `either` constraint in Example 2. Here, *equals* treats $X = t$ as a partial function and delays on the specialised `bound_t(X)`. Finally, *nonvar* implements the Boolean solver by delaying on the `nonvar(X)` ask-test (which holds if `X` is bound). Delaying on `nonvar` requires less delayed goals, since `nonvar(X)` subsumes both `X=t` and `X=f`. We believe an optimizing ask constraint compiler could translate *equals* to *nonvar* automatically.

Table 1(a) compares the execution times in milliseconds of the Boolean solvers on a test suite (details explained in [2]). Most of the overhead of *nonvar* compared to *hand* is due to the *nonvar* code retesting the `nonvar` ask-test

Prog	*hand*	*equals*	*nonvar*
`pigeon(8,7)`	524	988	618
`pigeon(24,24)`	157	338	167
`schur(13)`	7	11	5
`schur(14)`	57	87	65
`queens(18)`	4652	9333	5313
`mycie(4)`	1055	1988	1218
`fulladder(5)`	260	410	320
Geom. mean	237	179%	107%

(a)

Prog	*poly*	*mono*
`neq(10000)`	575	413
`neq(20000)`	1146	848
`neq(40000)`	2312	1676
`neq(80000)`	4592	3362
`square(4)`	414	219
`square(5)`	5563	2789
`square(6)`	2476	1213
`square(7)`	358	175
`square(8)`	12816	6168
`triples(3)`	88	70
`triples(4)`	535	436
`triples(5)`	4200	3526
Geom. mean	1349	64%

(b)

Table 1. Testing ask constraints: (a) Boolean benchmarks, (b) Sequence benchmarks

(which always holds if the retest predicate is woken up). The *equals* version adds overhead with respect to *nonvar* by using a greater number of (more specialised) ask constraints.

Our second experiment, shown in Table 1(b)), compares two versions of a sequence (Herbrand lists of finite domain integers) solver built using both a Herbrand solver for lists, and a finite domain (bounds propagation) solver. The resulting sequence solver provides three ask constraints over "complex" structures: `length(Xs,L)` (see Example 4), `append(Xs,Ys,Zs)` which constrains `Zs` to be the result of appending `Xs` and `Ys` (concatenation constraint), and `neq(Xs,Ys)` (see Example 5). The first benchmark `neq(`n`)` calls a single `neq(Xs,Ys)` constraint, then iteratively binds `Xs` and `Ys` to a list of length n (which eventually leads to failure). The second benchmark `square(`n`)` tries to find a $n \times n$ square of 1s and 0s such that no row/column/diagonal (in both directions) are equal. This is solved by first building the sequences for each row, column, diagonal and the reverse, then making each not equal to each other via the `neq` constraint, and then labeling. Here, `square(4)` has no solution, but `square(5-8)` do. The third benchmark `triples(`n`)` tries to find n triples of sequences of 1s and 0s such that (1) the length of each sequence is $\leq n$ (2) each sequence is not equal to any other sequence (from any triple); and (3) the concatenation for all triples must be equal. This example makes use of all three constraints, `length`, `append` and `neq`. All of `triples(3-5)` have solutions.

We use two versions of the sequence constraints. The first *poly* uses the polymorphic delay on equality for the `neq` constraints. The second, *mono* is a hand-edited version of *poly* where (1) all polymorphism has been specialised; and (2) a more efficient representation of the global id type is used. We can only use this more efficient global id type if we know in advance the types of the local ids, something not possible when using polymorphism. We see that, overall, *mono* is 36% faster than the more naïve *poly*.

Another interesting result is the linear behaviour of the `neq(`n`)` benchmarks with respect to n. As each list becomes more instantiated, we do not retest for equality over the entire list, rather we only retest the parts that have changed. If we retested the entire list, then we would expect quadratic behaviour.

8 Related Work and Conclusions

Ask constraints are closely related to dynamic scheduling, CHRs, reification and concurrent constraint programming.

The closest relation to this work is the `when` declarations of SICStus Prolog that allow a goal to delay until a test succeeds. These tests are similar to ask formulae (omitting existential quantification) over the primitive ask constraints `nonvar(X)`, `ground(X)` and `?=(X,Y)`. The last succeeds when X and Y are either known to be equal or not equal. The SICStus compiler appears to do much the same translation of conjunctions and disjunctions as defined in Section 4, but does not allow (explicit) existential quantification. The `?=(X,Y)` constraint includes the functionality of the ask equals defined in Section 6, but the SICStus implementation only deals with the a single constraint solver (Herbrand). The second difference is that the SICStus implementation does not break down testing of equality so that previous equal parts need not be retested.

CHRs are also closely related to this work, since an ask constraint is analogous to the guard of a CHR rule. We can consider the CHR rule (H`<=>`G`|`B) as equivalent to (H`<=>`(G`==>`B)) using ask constraints. This translation is generally inefficient, as delayed goals will be set up for every possible matching of H against the CHR store, and it is incompatible with some CHR optimisations, e.g. join ordering [3] and wakeup specialisation [2]. Instead, the guards for all rules are considered as a whole, and delayed goals are set up which may check multiple rules if a solver event occurs (see [2] for more details). Another difference is that non-Herbrand existential variables are not yet handled by any CHR implementation we are aware of, this remains future work.

Reified constraints allow similar functionality to ask constraints, particularly when combined with delaying an arbitrary goal until a Boolean variable is true. Both SICStus Prolog and ECLiPSe support reification of various constraints in their finite domain (and finite set) solvers, including conjunction, disjunction and implication. Again they do not handle explicit existential quantification.

One of the advantages of ask constraints over reification is they allow us to implement reified complex constraints which cannot be implemented using reification alone due to the interaction with existential quantifiers, as in Example 3. In that sense the ask construct is strictly more expressive than reification alone.

In both SICStus and ECLiPSe existential variables arising through normalization appear to be treated using the total function simplification described in Section 5, this can lead to erroneous behaviour. For example, in ECLiPSe the goal `ic:(Y < 0), ic:(B =:= (X + sqrt(Y) >= 2))` analogous to Example 13 incorrectly fails rather than set `B = 0`.

Guarded Horn Clauses [8] allows the programming of behaviour equivalent to ask formula for Herbrand constraints including conjunction, disjunction, and implicit existential quantifiers. The `cc(FD)` [9] language includes a blocking implication similar to ask constraints without commit, but only allows single constraints on the left hand side. However, one could use the `cardinality` constraint to mimic conjunction and disjunction. Both approaches treat a single solver, and do not handle explicit existential quantifiers.

Oz supports complex ask formula using constraint combinators [7]. Here ask constraints are executed in a separate constraint store which is checked for entailment by the original constraint store. This is a powerful approach which can handle examples that our approach cannot. However, its handling of existential variables is weaker than ours. For instance, the Oz equivalent to Example 13 will not set B to 1 when $X \geq 0$ and $Y \geq 1$. It would be interesting to extend Oz to handle existential variables better.

A constraint programming language supporting multiple solvers should support compilation of complex ask constraints. In this paper we have defined a solver-independent approach to this compilation, implemented it in HAL, and shown the resulting approach is practical and expressive. There is a significant amount of improvement that can be made to the naïve compilation strategy defined here, by transformations such as collecting calls for the same event. In the future we plan to investigate several optimizations.

References

1. M. García de la Banda, B. Demoen, K. Marriott, and P.J. Stuckey. To the gates of HAL: a HAL tutorial. In *Proceedings of the Sixth International Symposium on Functional and Logic Programming*, number 2441 in LNCS, pages 47–66. Springer-Verlag, 2002.
2. G.J. Duck, P.J. Stuckey, M. García de la Banda, and C. Holzbaur. Extending arbitrary solvers with constraint handling rules. In D. Miller, editor, *Proceedings of the Fifth ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*, pages 79–90. ACM Press, 2003.
3. 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.
4. J. Jaffar and J.-L. Lassez. Constraint logic programming. In *Proc. Fourteenth ACM Symp. Principles of Programming Languages*, pages 111–119. ACM Press, 1987.
5. J. Jaffar, M. Maher, K. Marriott, and P.J. Stuckey. The semantics of constraint logic programs. *Journal of Logic Programming*, 37(1–3):1–46, 1998.
6. V. Saraswat. *Concurrent Constraint Programming Languages*. PhD thesis, Carnegie-Mellon University, 1989.
7. C. Schulte. Programming deep concurrent constraint combinators. In *Practical Aspects of Declarative Languages (PADL 2000)*, volume 1753 of *LNCS*, pages 215–229. Springer, 2000.
8. K. Ueda. Guarded horn clauses. In E. Shapiro, editor, *Concurrent Prolog: Collected Papers*, pages 140–156. MIT Press, 1987.
9. P. Van Hentenryck, V. Saraswat, and Y. Deville. Design, implementation, and evaluation of the constraint language cc(FD). *Journal of Logic Programming*, 37(1–3):139–164, 1998.