Extending Arbitrary Solvers with Constraint Handling Rules

Gregory J. Duck
Peter J. Stuckey
Dept. of Computer Science
and Software Engineering
University of Melbourne, 3010
Australia
{gjd,pjs}@cs.mu.oz.au

Maria Garcia de la Banda
School of Computer Science
and Software Engineering
Monash University, 3800
Australia
mbanda@csse.monash.edu.au

Christian Holzbaur,
Dept. of Medical Cybernetics
and Artificial Intelligence
University of Vienna
Austria
christian@ai.univie.ac.at

ABSTRACT

Constraint Handling Rules (CHRs) are a high-level committed choice programming language commonly used to write constraint solvers. While the semantic basis of CHRs allows them to extend arbitrary underlying constraint solvers, in practice, all current implementations only extend Herbrand equation solvers. In this paper we show how to define CHR programs that extend arbitrary solvers and fully interact with them. In the process, we examine how to compile such programs to perform as little recomputation as possible, and describe how to build index structures for CHR constraints that are modified automatically when variables in the underlying solver change. We report on the implementation of these techniques in the HAL compiler, and give empirical results illustrating their benefits.

Categories and Subject Descriptors

D.3.2 [**Programming Languages**]: Language Classifications—*Constraint and logic languages*; D.3.3 [**Programming Languages**]: Language Constructs and Features—*Constraints*; D.3.4 [**Programming Languages**]: Processors—*Compilers*

General Terms

Languages, algorithms

Keywords

constraint handling rules, compilation, constraint solvers

1. INTRODUCTION

Constraint handling rules [6] (CHRs) are a flexible formalism for writing constraint solvers and other reactive systems. In effect, the rules define transitions from one constraint set to an equivalent constraint set, which serve to simplify constraints and detect satisfiability and unsatisfiability. CHRs have been used extensively (see e.g. [9]) and efficient implementations are already available for languages such as SICStus Prolog, Eclipse Prolog, Java [11], and HAL [4].

```
leq(A,A)             <=> true.
leq(A,B), leq(B,A)   <=> A = B.
leq(A,B) \ leq(A,B)  <=> true.
leq(A,B), leq(B,C)   ==> leq(A,C).
```

Figure 1: A CHR program for ordering constraints `leq` extending a Herbrand solver.

While the semantic basis of CHRs assumes an arbitrary underlying constraint system, in practice, implementations of CHRs only extend builtin Herbrand equation solvers. This means that the interactions between an underlying solver and the CHR system are not made explicit, but hidden in the implementation. In order to allow CHRs to extend arbitrary solvers, we need to examine exactly what these interactions are, and provide enough information to the CHR compiler to connect the two solvers. Let us illustrate the possible interactions with a simple example.

Example 1. Figure 1 shows a simple CHR solver which extends a Herbrand equation solver by defining the ordering constraint `leq`. The first rule states that if there exists a CHR constraint of the form `leq(A,A')` for which A and A' are identical, the constraint can be replaced by *true* since the ordering relation trivially holds. The second rule states that if there exist two CHR constraints with identical arguments A and B in opposite order, the two constraints can be replaced by the Herbrand constraint $A = B$. The third rule states that given two identical occurrences of a CHR constraint, we can replace one by *true*. Finally, the fourth rule states that given two CHR constraints of the form `leq(A,B)` and `leq(B',C)` where B and B' are identical, then we should add the transitive ordering relation `leq(A,C)`. Consider now the execution of goal `leq(X,Y), X = Y`. The execution will first add the CHR constraint `leq(X,Y)` to the store, which cannot by itself cause the application of any of the previous four rules. Once the Herbrand constraint `X = Y` is added to the Herbrand store, however, the first rule can be applied. □

We can see three kinds of interaction between the CHR

PPDP'03, August 27–29, 2003, Uppsala, Sweden.

solver and the underlying Herbrand solver in the example above.

- First, the CHR solver adds new constraints to the Herbrand solver, as in the second rule.
- Second, the CHR solver asks the Herbrand solver whether the equality of two terms is *entailed*, as in the first rule.
- Third, the Herbrand constraint solver must alert the CHR solver when changes in the Herbrand constraint store might cause entailment tests to succeed, as in the example goal. This is because the CHR rules then need to be reconsidered to check whether they can now be applied.

Constraint solvers, by definition, provide methods that allow new constraints to be added to their store. It is the last two kinds of interaction that need a well defined interface if we wish to extend an arbitrary underlying solver. Furthermore, in order to support efficient computation of CHR rules, indexes should be used to quickly detect possible rule firings [10]. If these indexes are based on solver variables, they depend on the underlying solver and, hence, the solver needs to communicate changes to the index.

The contributions of this paper are

- We show what features a solver must support in order to be extended by CHRs, and how we make these features visible to the HAL CHR compiler.
- We show how to reduce the amount of rule rechecking required by taking into account which changes in the underlying solver can cause a rule to fire.
- We show two methods to effectively build indexes for supporting CHR join computation over solver variables. Previous indexes were restricted to ground data.

The rest of the paper is organized as follows. In the next section we define CHRs together with their logical and operational semantics. Section 3 illustrates the basic compilation approach of CHRs. Section 4 shows how the CHR compiler can be informed about the capabilities of constraint solvers for entailment testing. In Section 5 we show how the compilation can minimize the amount of rechecking required, by better understanding how changes in the underlying solver store affect entailment testing. In Section 6 we discuss the use of indexes for CHRs that use solver variables. Section 7 presents the results of some experiments illustrating the benefits of specialized rechecking and indexes. Finally, Section 8 concludes.

2. CONSTRAINT HANDLING RULES

Constraint Handling Rules manipulate a global multiset of primitive CHR constraints (the CHR constraint store), using multiset rewrite rules which can take three forms

simplification	$c_1, \ldots, c_n \iff g \mid d_1, \ldots, d_m$	
propagation	$c_1, \ldots, c_n \implies g \mid d_1, \ldots, d_m$	
simpagation	$c_1, \ldots, c_l \setminus c_{l+1}, \ldots, c_n \iff g \mid d_1, \ldots, d_m$	

where the *head* $c_1, \ldots, c_n$ is a conjunction of CHR constraints, the *guard* g is a conjunction of constraints from the underlying solver, and the *body* $d_1, \ldots, d_m$ is a conjunction of CHR constraints and constraints of the underlying solver. Note that the guard is optional and, if omitted, it is equivalent to $g \equiv true$. A *CHR program* is a sequence of CHRs.

The simplification rule states that given a constraint store where the constraints in the underlying solver imply that there exists a multiset $\{c'_1, \ldots, c'_n\}$ in the CHR store which matches the multiset $\{c_1, \ldots, c_n\}$ and the guard g holds, then we can eliminate $\{c'_1, \ldots, c'_n\}$ from the CHR store and add the multiset $\{d_1, \ldots, d_m\}$. The propagation rule states that, for a matching constraint multiset $\{c'_1, \ldots, c'_n\}$ where g holds, we should add $\{d_1, \ldots, d_m\}$ to the store. The simpagation rules states that, given a matching constraint multiset $\{c'_1, \ldots, c'_n\}$ where g holds we can eliminate $\{c'_{l+1}, \ldots, c'_n\}$ from the store and add $\{d_1, \ldots, d_m\}$.

More formally, the logical interpretation of the rules is as follows.

$$\begin{array}{ll} \textit{simpl} & \forall \bar{x}(\exists \bar{y}\ g) \rightarrow (c_1 \wedge \cdots \wedge c_n \leftrightarrow (\exists \bar{z}\ d_1 \wedge \cdots \wedge d_m)) \\ \textit{propa} & \forall \bar{x}(\exists \bar{y}\ g) \rightarrow (c_1 \wedge \cdots \wedge c_n \rightarrow (\exists \bar{z}\ d_1 \wedge \cdots \wedge d_m)) \\ \textit{simpa} & \forall \bar{x}(\exists \bar{y}\ g) \rightarrow (c_1 \wedge \cdots \wedge c_n \leftrightarrow (\exists \bar{z}\ c_1 \wedge \cdots \wedge c_l \wedge \\ & \hfill d_1 \wedge \cdots \wedge d_m)) \end{array}$$

where $\bar{x}$ are the variables occurring in the head of the rule, $\bar{y}$ are the variables local to the guard, and $\bar{z}$ are the variables local to the body. We assume, for simplicity, that no local variables appear in both the guard and the body (i.e., that $\bar{y} \cap \bar{z} \subset \bar{x}$).

The operational semantics of CHRs exhaustively applies rules to the global multiset of constraints, being careful not to apply propagation rules twice on the same constraints (to avoid infinite derivations). It can be described as a transition system on a triple $\langle s, u, t \rangle_v$ where s represents the set of (numbered) CHR constraints currently in the CHR constraint store, u represents the conjunction of constraints currently in the underlying solver's store, t represents the set of (propagation rule) transitions that have already been applied, and v is a sequence of variables. The logical reading of $\langle s, u, t \rangle_v$ is as $\exists \bar{y}(s \wedge u)$ where $\bar{y}$ are the variables in the tuple not in v. The role of v is simply to record the universally quantified variables.

The transitions are defined as follows. Consider the tuple $\langle s, u, t \rangle_v$ and the (renamed apart) instance R of a propagation rule numbered a

$$c_1, \ldots, c_n \Longrightarrow_a g \mid d_1, \ldots, d_k, d_{k+1}, \ldots, d_m$$

where $d_1, \ldots, d_k$ are CHR constraints, and $d_{k+1}, \ldots, d_m$ are constraints of the underlying solver. Let us assume that there are numbered CHR constraints $\{c'_{i_1}, \ldots, c'_{i_n}\} \subseteq s$ such that $\models u \rightarrow \exists \bar{x}(c_1 = c'_{i_1} \wedge \cdots \wedge c_n = c'_{i_n} \wedge g)$, where $\bar{x}$ are the variables in R and there is no entry $(a, i_1, \ldots, i_n)$ in t. Then, a transition can be performed to give new state $\langle s \cup \{d_1, \ldots, d_k\}, u \wedge d_{k+1} \wedge \cdots \wedge d_m, t \cup \{(a, i_1, \ldots, i_n)\} \rangle_v$ where the new constraints added to s are given new id numbers.

The rule for simplification is simpler. Consider the tuple $\langle s, u, t \rangle_v$ and the (renamed apart) instance R of the simplification rule

$$c_1, \ldots, c_n \Longleftrightarrow g \mid d_1, \ldots, d_k, d_{k+1}, \ldots, d_m$$

where, $d_1, \ldots, d_k$ are CHR constraints, and $d_{k+1}, \ldots, d_m$ are constraints of the underlying solver. Let us assume that there are numbered CHR constraints $\{c'_{i_1}, \ldots, c'_{i_n}\} \subseteq s$ such that $\models u \rightarrow \exists \bar{x}(c_1 = c'_{i_1} \wedge \cdots \wedge c_n = c'_{i_n} \wedge g)$, where $\bar{x}$ are the variables in R. Then, the resulting tuple is $\langle s - \{c'_{i_1}, \ldots, c'_{i_n}\} \cup \{d_1, \ldots, d_k\}, u \wedge d_{k+1} \wedge \cdots \wedge d_m, t \rangle_v$, where, again, the new constraints added to s are given new

```
min(A,B,C)1                <=> A =< B | C = A.
min(A,B,C)2                <=> B =< A | C = B.
min(A,B,C)3                ==> C =< B, C =< A.
min(A,B,C)5 \ min(A,B,D)4 <=> C = D.
```

Figure 2: A simple min/3 solver based on extending a finite domain solver.

id numbers. The transition for a simpagation rule is defined analogously to the simplification rule. For more details see e.g. [1].

EXAMPLE 2. Consider the CHR program given in Figure 2 which extends an existing finite domain solver (`fdint`) that supports constraints of the form $A \leq B$ and $A = B$. The occurrences of `min`/3 are numbered for later reference. The new constraint `min(A,B,C)`, which is satisfied iff C is the minimum of A and B, is defined using four rules whose logical reading is as follows. The first rule (a simplification rule) states that if the finite domain solver can determine that $A \leq B$, then `min(A,B,C)` is equivalent to the finite domain constraint $C = A$. The second rule is similar but for the case $B \leq A$. The third rule (a propagation rule) states that if `min(A,B,C)` holds, then we require that the finite domain constraints $C \leq A$ and $C \leq B$ also hold. The last rule (a simpagation rule) encodes the functional dependency $min(A, B, C) \wedge min(A, B, D) \rightarrow C = D$ by stating that if both `min(A,B,C)` and `min(A,B,D)` hold, this is equivalent to having only one of them hold and the finite domain constraint $C = D$.

The operational reading of these rules is as follows. The first rule can be applied if there is a constraint `min(A,B,C)` in the CHR store for which the finite domain solver can determine that $A \leq B$. In that case, it will eliminate `min(A,B,C)` from the CHR store and add the finite domain constraint $C = A$ to the FD store. Similarly for the second rule. The third rule can be applied for any constraint `min(A,B,C)` in the CHR store with identifier i for which the tuple $(3, i) \notin t$. If so, it will add the finite domain constraints $C \leq A$ and $C \leq B$ to the FD store, and the tuple $(3, i)$ to t. Finally, the fourth rule can be applied if there are two constraints of the form `min(A,B,C)` and `min(A,B,D)` in the CHR store. If so, it will remove the second one and add the finite domain constraint $C = D$ to the FD store. □

3. BASIC COMPILATION OF CHRS

In this section we recall how CHRs are compiled to logic programs, see [8, 10] for more details. Compilation starts with a preprocessing step to make all guards explicit, since non-variable terms and matching variables appearing in the head of a rule actually indicate guards. Normalization is achieved by iteratively applying the following steps

1. If variable X appears more than once in the head of a rule, replace one location with a new variable, say X', and add the constraint $X = X'$ to the guard.

2. If any argument of any CHR constraint in the head of a rule is not a variable (say some term c), then replace with a new variable, say X', and add the constraint $X' = c$ to the guard

After normalization, each head simply provides the multiset of names of constraints which can match that rule, while the guard indicates which such multisets actually match.

```
leq(A,A')1              <=> A = A'          | true.
leq(A,B)3, leq(B',A')2  <=> A = A', B = B'  | A = B.
leq(A,B)5 \ leq(A',B')4 <=> A = A', B = B'  | true.
leq(A,B)7, leq(B',C)6   ==> B = B'          | leq(A,C).
```

Figure 3: Example CHR program for ordering constraints leq with explicit guards.

EXAMPLE 3. Consider the CHR program in Figure 1. Figure 3 shows the result of normalizing this program. We have also numbered each occurrence of `leq`/2 in the head of a rule; this will be used later in the compilation process. □

After normalization is performed, the compilation of CHRs needs to produce code that determines when to fire rules. The operational semantics introduced in the previous section is highly abstract, leaving such question unresolved. In practice, execution proceeds as follows. Every time a new CHR constraint (the *active constraint*) c is placed in the store, we search for a rule that can now fire thanks to c, i.e., a rule for which there is now a multiset of CHR constraints (including c) in the CHR store that matches its head and causes the guard to hold. The first (in the textual order they appear in the program) such rule R found is fired. If c has not been deleted by R, we continue checking for more rules that can make use of c.

The above method is usually implemented by associating code to each occurrence of a constraint in the head of a rule R (i.e., each c_i in R). The code will be executed if the active constraint c matches the associated occurrence c_i. The aim of the code is to search for partner constraints (i.e., constraints in the store that match other $c_j, j \neq i$ in the head of R) that, together with c, will entail the guard and cause R to fire. Compiling CHRs thus consists of creating code for each occurrence c_i in the program, and joining these in textual order.

There are two kinds of search for partners: *existential search* which looks for a single set of partners and *universal search* which looks for all possible partners. The former is used if the active constraint gets deleted by R since, as soon as we find the first set of partners, R fires and the active constraint is deleted making it impossible for R to fire again. The latter is used if the active constraint does not get deleted by the rule since then the same rule can fire multiple times, one for each set of partners found.

EXAMPLE 4. Figure 4 shows the (simplified) code produced by the compilation of the normalized CHR program shown in Figure 3. When a new active `leq(X,Y)` constraint is executed, it is first given a new identifier by `new_id(Id)`, then it is added to the store (using `insert(X,Y,Id)`).[1], Next some delay goals are set up (by `delay_leq(X,Y,Id)`) for rechecking the active constraint, we defer discussion of how this works until Section 5. Finally, the code `leq_`i`(X,Y,Id)` associated to each `leq(X,Y)`$_i$ in Figure 3, is called in turn.

The reader should feel free to skip the rest of the description of the code in Figure 4. It gives a reasonably detailed understanding of the CHR compilation process, not all of

[1]For simplicity, we give an implementation of the CHR constraint store using a Prolog database. This will not work correctly for non-ground constraints, (since the variables identities are not maintained) but illustrates concisely the meaning of the predicates for store manipulation. In practice, lists of constraints are stored internally.

```
leq(X,Y) :- new_id(Id), insert_leq(X,Y,Id),
   delay_leq(X,Y,Id), leq(X,Y,Id).

leq(X,Y,Id) :-
   leq_1(X,Y,Id), leq_2(X,Y,Id), leq_3(X,Y,Id),
   leq_4(X,Y,Id), leq_5(X,Y,Id), leq_6(X,Y,Id).
   leq_7(X,Y,Id).

leq_1(X,Y,Id) :- (alive(Id), X == Y -> delete(Id) ; true).
leq_2(X,Y,Id) :- (alive(Id),
   a_leq(A,B,Id2), Y == A, X == B, Id ≠ Id2 ->
      delete(Id), delete(Id2), X = Y
   ; true).
leq_3(X,Y,Id) :- leq_2(X,Y,Id).
leq_4(X,Y,Id) :-
   (alive(Id),a_leq(A,B,Id2), X == A, Y == B, Id ≠ Id2 ->
      delete(Id)
   ; true).
leq_5(X,Y,Id) :- all_leq(L), leq_5a(L,X,Y,Id).
leq_5a([],_,_,_).
leq_5a([leq(A',B',Id2)|L],X,Y,Id) :-
   (X == A', Y == B', alive(Id), alive(Id2) -> delete(Id2)
   ; true ),
   leq_5a(L,X,Y,Id).
leq_6(X,Y,Id) :- all_leq(L), leq_6a(L,X,Y,Id).
leq_6a([],_,_,_).
leq_6a([leq(A,B,Id2)|L],X,Y,Id) :-
   (B==X, \+ fired(leq,4,Id2,Id),alive(Id),alive(Id2) ->
      assert(fired(leq,4,Id2,Id)), leq(A,Y)
   ; true ),
   leq_6a(L,X,Y,Id).
leq_7(X,Y,Id) :- all_leq(L), leq_7a(L,X,Y,Id).
leq_7a([],_,_,_).
leq_7a([leq(B',C,Id2)|L],X,Y,Id) :-
   (B'==Y, \+ fired(leq,4,Id,Id2),alive(Id),alive(Id2) ->
      assert(fired(leq,4,Id,Id2)), leq(X,C)
   ; true ),
   leq_7a(L,X,Y,Id).

alive(Id) :- sleq(_,_,Id).
delete(Id) :- retract(sleq(_,_,Id)).
a_leq(X,Y,Id) :- sleq(X,Y,Id).
all_leq(L) :- findall(leq(A,B,Id2),sleq(A,B,Id2),L).
insert_leq(X,Y,Id) :- assert(sleq(X,Y,Id)).
```

Figure 4: Compiled code for a new active `leq` constraint.

which is required to understand the rest of the paper. For the curious reader, here is an explanation of the code.

The code associated to the first occurrence first checks that the constraint is still alive. This is in fact needed by the code of every occurrence since the occurrence could have been removed at some point before executing the code. If the occurrence is indeed alive it checks whether the guard is satisfied (the two arguments are in fact the same variable), in which case it deletes the constraint.

The code associated to the second occurrence checks the active constraint is still alive and, if so, it searches for a stored `leq` constraint that (a) matches occurrence 3, (b) has the same arguments as the second occurrence in the opposite order, and (c) is not the active constraint (i.e., it has a different identifier). If such a constraint exists, then both constraints are deleted, and the body of the rule `A = B` is executed. This is an example of existential search. The code for the third occurrence is identical to that of the second occurrence.

The code for the fourth occurrence is similar to that of the second, except the arguments should appear in the same order, and only the active constraint is deleted. The code for the fifth occurrence requires universal search since the active constraint will not be deleted. In doing this it collects all the stored `leq` constraints and iterates through them one by one looking for matching partners which are then deleted.

The code for the sixth occurrence is the first example of a propagation rule. Like the code for occurrence five, it collects all the stored `leq` constraints, and iterates through them looking for a match. It also checks that this rule has not fired before using the same CHR constraints. Finally, just before firing, it checks that both constraints are still alive. If so, it records information about the firing and then executes the body of the rule. The seventh occurrence is similar. □

The code given in Figure 4 has been simplified for ease of explanation. In practice, many optimizations are also applied (see [10] for details) to improve efficiency. For example, the code for occurrence 3 can be removed since it is identical to that of occurrence 2. The code for occurrence 5 can also be removed since it will never fire (occurrence 4 will always remove the active constraint first). Furthermore, the code could postpone adding the `leq` constraint to the store (in case it is going to be removed immediately after), and remove unnecessary checks for liveness (see Example 13 later).

The operational semantics illustrated above glossed over two important points related to the interaction of CHRs with arbitrary constraints solvers: the need for guard entailment testing and the need to reconsider CHR rules to check whether the guard is now entailed. These issues will be discussed in the next two sections.

4. GUARD ENTAILMENT TESTING

Let R be a normalized CHR rule with guard g. The operational semantics of CHRs dictates that R can only fire iff g is entailed by the matching and the current state u of the underlying solver. In practice, this means that the underlying solver must not only provide a procedure for *telling* a constraint (adding it to the underlying constraint store) whenever it appears in the body of the rule, but also a procedure for *asking* a constraint (determining if the guard is entailed by the current underlying constraint store) whenever it appears in the guard of the rule. For example, in the case of the Herbrand solver used previously, the only tell constraint (`=/2`) has the known associated ask constraint `==/2`.

Solvers in HAL must define a type for the solver variables (e.g., `fdint`), and code to initialize new solver variables. Often a solver variable will be some form of pointer into a global store of variable information. The solver also defines predicates for the constraints supported by the solver. These predicates define *tell* constraints for that solver (e.g., they provide the code for predicate `X =< Y` which adds constraint $X \leq Y$ to the solvers store). Often the solver is also packaged as an appropriate instance of a solver type class, and thus the solver is known to provide at least the constraints included in the type class interface. For more details on HAL solver classes see e.g., [3].

In order for a constraint solver to be extended by CHRs, the solver needs to provide *ask* versions of the constraints that it supports. It is the ask version of the constraints that should be used in guards. For example, the first rule of the `min` program shown in Figure 2 should be re-written as

```
min(A,B,C) <=> 'ask_=<'(A,B) | C = A.
```

where `ask_=<` is the ask version of the finite domain `=<` constraint.

This transformation is performed automatically by the compiler because

- this removes the burden from the programmer of understanding the relationships between tell and ask constraints, thus reducing the number of programmer errors; and
- the original rule is far simpler and more aesthetic, capturing the programmers intentions more clearly.

In fact, such automatic transformation is performed by every CHR implementation we know of. However, these implementations only deal with one underlying constraint solver (Herbrand). When arbitrary solvers are used, the compiler needs a general method for determining the relationship between the tell and ask versions of each constraint so that it can automatically transform one into the other. In HAL this is achieved by the following `asks` declaration.

:- <*ask-constraint*> `asks` <*tell-constraint*>.

which defines a mapping from a tell to an ask constraint. Hence, the finite domain solver `fdint` might declare

```
:- 'ask_=<'(X,Y) asks X =< Y.
:- X == Y asks X = Y.
```

The `asks` declaration is effectively a macro definition on which the following restrictions apply. Each tell constraint can only have one associated ask constraint (although an ask constraint can be associated to more than one tell). The arguments of the *tell-constraint* must be distinct variables, and only these and anonymous variables can appear in the corresponding *ask-constraint*. And finally, the ask constraint must be defined for the type of arguments of the associated tell constraints, it must be usable in any mode in which the associated tell constraints are, and it should either succeed once or fail.

Example 5. The result of compiling the CHR program of Figure 2 in the presence of the two ask declarations included above, is shown in Figure 5. Note the transformation of the guard constraints. □

A predicate is recognized by the compiler as a tell constraint iff it has been declared as having an associated ask constraint. The compiler automatically replaces each such tell constraint which textually appears in a guard with its ask version. HAL (and Prolog CHR implementations) also allow arbitrary predicates in the guard. This means that tell constraints nested inside the guard will be treated as tells, when perhaps this was not the intention of the programmer. The HAL compiler warns if this can occur.[2]

The need for connecting tell constraints with ask constraints has been recognised before. SICstus Prolog allows finite domain constraints defined by indexicals to be defined with an attached ask version.

It is sometimes possible to provide useful default implementations of ask constraints. For example, when all arguments are ground the ask constraint and tell constraint are known to be equivalent. Thus, for solvers that support the notion of a fixed variable (i.e., can check whether a variable is fixed to a single value) we can provide a useful default ask constraint defined as follows:

```
:- ask_p(A1,...,An) asks p(A1,...,An).
ask_p(A1,...,An):- val(A1,_), ..., val(An,_), p(A1,...,An).
```

where the call `val(Var,Val)` succeeds if solver variable `Var` has the fixed value `Val`.[3] We plan to extend the solver type classes [7] provided by HAL to include `asks` declarations, which will make it possible to automate the construction of ask defaults using the default class methods of HAL.

```
min(X,Y,Z) :- new_id(Id), insert_min(X,Y,Z,Id),
   delay_min(X,Y,Z,Id), min(X,Y,Z,Id).
min(X,Y,Z,Id) :-
   min_1(X,Y,Z,Id), min_2(X,Y,Z,Id),
   min_3(X,Y,Z,Id), min_4(X,Y,Z,Id), min_5(X,Y,Z,Id).
min_1(X,Y,Z,Id) :-
   (alive(Id), 'ask_=<'(X,Y) -> delete(Id), Z = X ; true).
min_2(X,Y,Z,Id) :-
   (alive(Id), 'ask_=<'(Y,X) -> delete(Id), Z = Y ; true).
min_3(X,Y,Z,Id) :-
   (\+ fired(min,3,Id),alive(Id) ->
      assert(fired(min,3,Id)), Z =< X, Z =< Y ; true).
min_4(X,Y,Z,Id) :-
   (alive(Id), a_min(A,B,C,Id2), alive(Id2),
      A == X, B == Y -> delete(Id), C = Z
   ;  true).
min_5(X,Y,Z,Id) :- all_min(L), min_5a(L,X,Y,Z,Id).
min_5a([],_,_,_,_).
min_5a([min(A,B,C,Id2)|L],X,Y,Id) :-
   (X == A, Y == B, alive(Id), alive(Id2) ->
      delete(Id2), Z = C ; true),
   min_5a(L,X,Y,Z,Id).
```

Figure 5: Code for executing an active `min/3` constraint.

5. RECHECKING RULES

An ask constraint succeeds if the underlying solver can prove that the constraint is entailed by the current store. Changes in the constraint solver can make an ask constraint succeed when it previously failed. Hence, when the underlying solver store changes we should revisit the CHR rules to see if they can now fire. This requires the CHR compiled code to establish a connection between the CHR constraints and the underlying solver so that the CHR constraints can be re-checked when necessary.

Current implementations of CHRs only extend the Herbrand solver and manage re-checking of constraints as follows. Anytime a variable occurring in a CHR constraint is modified (and this could just mean that a variable in the term is bound to another variable occurring in a CHR), then that CHR constraint is rechecked.

Example 6. The (missing) definition of the predicate `delay_leq(X,Y,Id)` from Figure 4 finds all the variables occurring in the term `leq(X,Y)` and attaches a delay goal to

[2]Since Prolog implementations only interact with the Herbrand solver, there is a devious technique of marking variables appearing in the guard, to check that no guard variable is modified. This effectively at runtime converts Herbrand tell constraints to ask constraints. It does not however notice or convert constraints for other solvers, such as SICStus Prologs built in `clpfd` finite domain solver. Hence, it can give equally erroneous behavior.

[3]For example, the value returned by `val` on a fixed `fdint` variable will be an integer (`int`).

each variable so that when it is modified the goal leq(X,Y,Id) is executed, thus treating the constraint as active once more and checking for any possible partners. Note that by re-executing leq(X,Y,Id) instead of the original constraint leq(X,Y) the re-activated constraint (a) is not added (again) to the store, and (b) keeps the original identification number, thus preventing it from firing propagation rules twice. □

The above solution works because Prolog CHRs only extend the Herbrand solver. To extend arbitrary solvers a more general method is needed. In this section we discuss the approach implemented within the HAL compiler.

HAL allows the creation of solvers that support dynamic scheduling [3]. A dynamically scheduled construct has the form

$cond_1$ ==> $goal_1$ || ... || $cond_n$ ==> $goal_n$

where each wake condition $cond_i$ represents some sort of solver event, such as "the variable X has been touched" "the lower bound of X has changed", etc, and $goal_i$ is a HAL goal. Here, the token '==>' should not be confused with the CHR propagation arrow.[4]

Example 7. Consider our example fdint finite domain solver introduced in Example 2. The wake conditions supported by this solver (the usual ones for a finite domain solver) are

fixed(X)	the domain of X is reduced to a single value.
lbc(X)	the lower bound of X changes (increases).
ubc(X)	the upper bound of X changes (decreases).
dc(X)	the domain of X changes (reduces).

Note that these conditions are not mutually exclusive. For example, if the domain of X changes from $\{1, 3, 5\}$ to $\{1\}$, then the conditions fixed(X), ubc(X) and dc(X) all hold. □

The delay construct introduced above causes each goal $goal_i$ to be (re)executed every time the wake condition $cond_i$ holds. Note that, by default, a wake condition can become true multiple times and that, therefore, the associated goal will be re-executed every time this happens. HAL associates a unique identifier with every delay construct and allows the user to "kill" the entire construct by including the literal kill inside any of the $goal_i$. This is useful, whenever the success of a wake condition means there is no further benefit in re-executing the other goals. More information about HAL's dynamic scheduling interface can be found in [3].

We can use the above dynamic scheduling construct to reconsider CHR constraints whenever changes in the store of the underlying solver make it necessary.

Example 8. Consider the compiled code of Figure 5 and the fdint finite domain solver. We can use the dc condition introduced in the previous example to implement a naïve delay_min predicate which will recheck the min constraint any time one of the domains of its variables changes.

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

Then, the execution of goal

[4]Dynamic scheduling syntax existed in HAL before CHRs were added.

```
[X,Y,Z] in 0..9, min(X,Y,Z), Z ≠ 2, Y ≤ 3, X ≥ 5.
```

proceeds as follows. First, [X,Y,Z] in 0..9 is added to the fdint store, causing the domains of the variables X, Y and Z to be set to $\{0..9\}$. Next, min(X,Y,Z,1) is added to the CHR store, its delay goals are set up, and its occurrences are checked. In checking its occurrences only occurrence three (third rule) causes the rule to fire, resulting in the two fdint constraints $Z \leq X$ and $Z \leq Y$ being added to the fdint store. Neither of these constraints changes the domains of the variables and, therefore, min(X,Y,Z,1) is not re-executed. When the next constraint $Z \neq 2$ is added to the fdint store, the domain of Z becomes $\{0..1, 3..9\}$. Since the finite domain of Z has changed, min(X,Y,Z,1) is re-executed. Again, only the third rule can fire, but since it has already fired for $Id = 1$, the rule is not reapplied. We then add $Y \leq 3$ to the fdint store, which changes the domain of Y to $\{0..3\}$ and that of Z to $\{0..1, 3\}$. This causes min(X,Y,Z,1) to be re-executed twice and, again, no new rule is fired. Finally, $X \geq 5$ is added changing the domain of X to $\{5..9\}$. This causes min(X,Y,Z,1) to be re-executed, and the second rule to fire since $Y \leq X$ now holds. Thus min(X,Y,Z,1) is deleted from the store and the constraint $Z = Y$ is added to the fdint store changing the domain of Y to $\{0..1, 3\}$. This causes min(X,Y,Z,1) to be re-executed but since the constraint is no longer alive the execution terminates. □

The naïve re-execution above rechecks every possible rule every time the solver state for the variables involved changes. Often this causes no new rules to fire. We can improve upon this by (a) determining a set of wake conditions which accurately signal the possible entailment of an ask constraint, and (b) building code that only reconsiders occurrences associated to those ask constraints.

Example 9. Consider the following implementation of the ask '≤' constraint for the fdint finite domain solver:

```
'ask_=<'(X,Y) :-
   UBX = fd_max(X), %% get current X upper bound
   LBY = fd_min(Y), %% get current Y lower bound
   UBX =< LBY.      %% integer comparison
```

where functions fd_max/1 and fd_min/1, respectively, return the (integer) upper and lower bounds of a variable's current domain. Note that this is an incomplete test since, even if the constraint $X \leq Y$ has been added to the store, the ask constraint may not succeed. Given this definition, the answer to the ask constraint will only change if the wake conditions ubc(X) or lbc(Y) becomes true. Other possible conditions will never signal a possible change in the answer to the ask constraint, unless one of these conditions also occurs.

Consider the following implementation of the ask '=' constraint for the fdint solver:

```
X == Y :- val(X,Value), val(Y,Value).
```

The ask constraint for $X = Y$ (X == Y) will only succeed if X and Y are both fixed to the same value. As in most finite domain solvers the implementation of the ask '=' constraint is quite incomplete. Given this definition, the only wake conditions where we should re-check a $X = Y$ guard are fixed(X) or fixed(Y).

A list of finite domain ask constraints and the corresponding wake conditions is given in Figure (6). $X \geq Y$ ask

Constraint	*Conditions*
$X = Y$	fixed(X), fixed(Y)
$X \neq Y$	dc(X), dc(Y)
$X \leq Y$	ubc(X), lbc(Y)
$X \geq Y$	lbc(X), ubc(Y)

Figure 6: Relationship between finite domain ask constraints and wake conditions.

constraints may change answers on `lbc(X)` or `ubc(Y)`, the opposite for $\leq$ constraints. Disequality ask constraints ($\neq$) may change answers on any generic domain change. □

Even in the case of a Herbrand solver we can do better than examine every rule for each CHR constraint whenever a variable in that CHR is bound (perhaps to another variable).

EXAMPLE 10. The HAL Herbrand solver supports two wake conditions: `touched(X)` if the variable X is touched (bound to another variable or a structure), and `bound(X)` if the variable X is bound to a structure. An ask constraint of the form $X = Y$ must be revisited if either X or Y are touched, while an ask constraint of the form $X = f(Y)$ need only be revisited if X is bound.

The `leq`/2 example does not include any ask constraints with structure, hence this provides no improvement. But we can still improve the naïve wakeup of the Prolog methodology by noticing that, for occurrence 6, only changes in the first argument can signal the entailment of its guard, since the second argument does not appear in it. Similarly occurrence 7 only depends on the second argument. □

In order to allow each solver to provide a list of the relevant wake conditions for each ask constraint, We extend the `asks` declaration (introduced in the previous section) to allow each solver to provide a list of the relevant wake conditions for each ask constraint, as follows

```
:- <ask-constraint> asks <tell-constraint>
                    [ wakes <wake-condition>* ].
```

The first part of the declaration is the same as before, where a mapping between an ask and tell constraint is defined. The new part, prefixed by token '`wakes`', provides a list of wake conditions that may cause the ask constraint to succeed. We shall refer to this as the *wakes list*. The wakes list is optional, and by default it will be empty.[5] The HAL compiler uses type analysis to ensure that each wake condition is supported by the solver.

EXAMPLE 11. Our finite domain solver `fdint` provides the following declarations:

```
:- 'ask_=<'(X,Y) asks X =< Y wakes [ubc(X),lbc(Y)].
:- X == Y asks X = Y wakes [fixed(X),fixed(Y)].
```

indicating that the ask `X =< Y` constraint only needs to be re-checked whenever the `ubc(X)` or `lbc(Y)` conditions become true. Similarly, the ask `X = Y` constraint only needs to be re-checked whenever the `fixed(X)` or `fixed(Y)` conditions become true. □

[5]An empty wakes list implies no wake condition affects the ask constraint.

```
delay_min(X,Y,Z,ID) :-
   (  ubc(X)   ==> ubc_X_min(X,Y,Z,ID)
   || ubc(Y)   ==> ubc_Y_min(X,Y,Z,ID)
   || lbc(X)   ==> lbc_X_min(X,Y,Z,ID)
   || lbc(Y)   ==> lbc_Y_min(X,Y,Z,ID)
   || fixed(X) ==> fixed_X_min(X,Y,Z,ID)
   || fixed(Y) ==> fixed_Y_min(X,Y,Z,ID)
   ).

ubc_X_min(X,Y,Z,ID) :- min_1(X,Y,Z,ID).
ubc_Y_min(X,Y,Z,ID) :- min_2(X,Y,Z,ID).
lbc_X_min(X,Y,Z,ID) :- min_2(X,Y,Z,ID).
lbc_Y_min(X,Y,Z,ID) :- min_1(X,Y,Z,ID).
fixed_X_min(X,Y,Z,ID) :- min_4(X,Y,Z,ID), min_5(X,Y,Z,ID).
fixed_Y_min(X,Y,Z,ID) :- min_4(X,Y,Z,ID), min_5(X,Y,Z,ID).
```

Figure 7: Optimized compiled `min`/3 delay and wakeup handling code.

The extended `asks` declarations allow the HAL CHR compiler to determine more accurately which occurrences need to be re-checked for which wake conditions. In order to do this, the compiler examines every possible wake condition for each variable, and determines the subset of occurrences that must be examined if a wake condition become true. It then produces specialized code for each wake condition.

EXAMPLE 12. For the constraint `min(X,Y,Z,Id)` if the upper bound of X changes (`ubc(X)`), only the occurrence in the first rule needs to be re-checked because of the guard $X \leq Y$. No other guard is affected by the `ubc(X)` condition. Similarly, if variable X becomes fixed (`fixed(X)`) then only occurrences 4 and 5 from the last rule need to be re-checked, since the guard for this rule contains equality constraints. The resulting optimized implementation of `delay_min` is shown in Figure 7.

Each wake condition causes the execution of a specialized wakeup predicate. The wakeup predicate `ubc_X_min` for condition `ubc(X)` only re-checks occurrence `min_1` from the first rule. Similarly, the wakeup predicate `fixed_X_min` for condition `fixed(X)` only re-checks occurrences `min_4` and `min_5`. Notice that for some conditions, for example `dc(Z)`, no occurrences ever need to be re-checked. This is obviously true, since variable Z (after renaming as C or D) never appears in any of the guards. In this case the compiler can completely eliminate all consideration of that condition.

Using this version of `delay_min` the execution of the goal

`[X,Y,Z] in 0..9, min(X,Y,Z), Z` $\neq$ `2, Y` $\leq$ `3, X` $\geq$ `5.`

proceeds as follows. The first constraint sets the domains as before. Next the `min(X,Y,Z,1)` is added, the delay goal set up, and each rule checked. As before, the third rule fires adding the constraints $Z \leq X$ and $Z \leq Y$ which do not change any domains. Now, when we add $Z \neq 2$ to the `fdint` store, the domain of Z changes. However, no guard mentions Z and therefore there is no delayed goal that wakes when Z's domain changes. When we add $Y \leq 3$ to the `fdint` store, only the upper bounds of Y and Z change. This causes the goal `ubc_Y_min` to execute, which checks the second occurrence of `min` only. Similarly, when the constraint $X \geq 5$ is added to the `fdint` store, the lower bound of X changes and, again, the only the second occurrence of `min` is checked. This time the rule fires, the CHR constraint is deleted and the constraint $Z = Y$ added to the store. This version makes 7 occurrence checks, rather than the 22 performed by the naïve version. □

Clearly, it is generally desirable to avoid re-checking occurrences which we know are still doomed to fail, since this avoids redundant work. The optimized re-execution can be *arbitrarily* faster than the naïve approach since checking an occurrence could be arbitrarily difficult.

There are however tradeoffs in creating the specialized delay code. In order to avoid code explosion, the compiler creates a separate predicate for each individual occurrence. This means we can straightforwardly create the different sequences of occurrences that are required for each wakeup condition. Previous CHR compilers for HAL and SICStus chained the code for each predicate to the next occurrence, and performed optimizations that relied on this fixed order. For instance, it was possible to avoid checks for liveness of the active CHR constraint by never executing code for occurrences after firing a rule where the active constraint is deleted.

EXAMPLE 13. If we chain the occurrences for `min` together we can remove unnecessary aliveness checks and not check any occurrences after the active constraint is deleted. A fragment of the chained code is shown below:

```
min(X,Y,Z) :- new_id(Id), min_1(X,Y,Z,Id).
min_1(X,Y,Z,Id) :-
   ('ask_=<'(X,Y) -> delete(Id), Z = X ; min_2(X,Y,Z,Id)).
min_2(X,Y,Z,Id) :-
   ('ask_=<'(Y,X) -> delete(Id), Z = Y ; min_3(X,Y,Z,Id)).
min_3(X,Y,Z,Id) :-
   Z =< X, Z =< Y, min_4(X,Y,Z,Id).
...
```

Clearly we cannot use this code for the wakeup predicates. □

We could choose to separately optimize each of the sequences of occurrences that occur in each wakeup predicate, and suffer the resultant code size increase, but at present the HAL compiler does not.

In creating a wakes list, the solver writer should endeavour to use a *complete* set of wakeup conditions, so that any change in the solver state captured by a wake condition which could cause the ask constraint to succeed is discovered. Failure to provide a complete list will result in some rules not being re-checked when they could now succeed. Hence, the CHR store might not be as reduced as it otherwise should be. Note that this is still correct, in the sense that CHR rules only change the constraint store to a logically equivalent store, so the original store is still a correct answer. Incompleteness of the wakes list thus leads to incompleteness of the CHR extension, but this often simply reflects the incompleteness of the tell and ask constraints as implemented by the underlying solver.

The solver writer should also endeavour to use a *minimal* set of wakeup conditions in order to generate more efficient code. When a wakes list is not minimal the resulting delayed goals may be called more often than required.

EXAMPLE 14. For example, the following `asks` declaration is complete, but not minimal.

```
:- X == Y asks X = Y wakes [dc(X),fixed(X),fixed(Y)].
```

The domain change conditions `dc(X)` is redundant, since the conditions `fixed(X)` and `fixed(Y)` are already enough to cover ask $X = Y$. Given this declaration the CHR compiler will generate redundant specialized re-activation predicates that check any =/2 guard on `dc(X)`, causing occurrences to be checked when there is no need. □

For solver classes which support delay on fixed values we can provide the default `asks` declaration

```
:- ask_p(A1,...,An) asks p(A1,...,An)
                         wakes [fixed(A1),...,fixed(An)].
```

As described, CHRs can only (usefully) extend underlying solvers that support delay, since otherwise CHR rules are never revisited. This may appear to be a strong restriction, since many external solvers will not support any wake conditions. There are at least two solutions to this problem that do not require modifying the solver code itself.

The simplest is a programming solution. By attaching an artificial constraint to each CHR rule we can trigger the rechecking of all rules, by reinserting a new copy of the constraint.

EXAMPLE 15. If the `fdint` solver does not support dynamic scheduling, then rewriting the `min` solver as:

```
redo \ min(A,B,C)             <=> A =< B | C = A.
redo \ min(A,B,C)             <=> B =< A | C = B.
redo, min(A,B,C)              ==> C =< B, C =< A.
redo, min(A,B,C) \ min(A,B,D) <=> C = D.
```

means that every time we add a `redo` constraint, all rules will be rechecked. □

The above programming solution is extremely expensive since it rechecks every possible rule and has no understanding of changes in the underlying solver state. If the solver provides reflection predicates that allow the user to see the internal state there is a better solution. We can build a "wrapper" solver around the underlying solver that attaches to each underlying solver variable its previous (internal) state. The wrapper solver can then implement (not very eager) dynamic scheduling by periodically checking which variables have changed state in the underlying solver.

EXAMPLE 16. Let us assume that the `fdint` solver does not support dynamic scheduling, but does allow the current domain D of a variable X to be returned (using `fd_dom(X,D)`). Then, the following psuedo-code checks that the `dc` "domain change" condition holds and if so stores the current domain, as the "last considered" domain. Note it (implicitly) uses a global index to store domain information attached to each solver variable.

```
check_dc(X) :-
   prev_domain(X,D0), %% get previous stored domain
   fd_dom(X,D),       %% get current domain from fdint
   D ≠ D0,            %% if different domain has changed
   store_domain(X,D). %% replace old stored domain
```

□

6. BUILDING INDEXES ON SOLVER VARIABLES

In [10] we showed that building indexes for lookups dramatically improves the time performance of CHR programs. This is not surprising, since we can find matching constraints for rule heads much more efficiently. The first version of the HAL CHR compiler described in [10] was restricted to CHRs which did not extend any solver, hence it generated efficient indexes for ground data. In this section we examine how to build efficient index structures on data involving solver variables, a task complicated by the need to take the solver state into consideration.

As previously mentioned, the main task of the compiled code associated to each occurrence of a CHR constraint in rule R is to find partner constraints that cause R to fire. This is a (possibly complicated) *relational join* operation.

EXAMPLE 17. Consider the code for occurrence 6 in

$\texttt{leq(A,B)}_7\texttt{, leq(B,C)}_6\texttt{ ==> leq(A,C).}$

Given an active CHR constraint `leq(X,Y)` matching occurrence 6, we need to find CHR constraints of the form `leq(_,X)`. We could quickly determine the possible partners if we had stored all the `leq` CHR constraints in an index on their second argument. □

Other CHR systems do not make use of indexes in calculating the join. The basic approach to determining join partners in the Prolog implementations we are aware of, is close to the Prolog database solution illustrated in Figure 4. All CHR constraints with the same functor and arity p/n are stored in a list, which is iterated over when performing an existential or universal search.

However, these Prolog implementations do provide one lookup mechanism (using attributed variables) which improves the search for partners for non-ground CHR constraints. The mechanism consists of attaching to each variable X a list for each CHR constraint p/n, of constraints which contain X. For example, a list of `min/3` constraints containing X, and a list for `leq/2` constraints containing X. In order to look for a partner constraint p/n, one selects a variable X occurring in the matching argument, finds the list of constraints attached to X for p/n, and searches for one with the required form. This provides very efficient lookups when arguments are single variables, but poor lookups otherwise. Furthermore, the technique is not applicable when the argument is ground.

EXAMPLE 18. The attributed variable approach to partner search of occurrence 6 in the rule

$\texttt{leq(A,B)}_7\texttt{, leq(B,C)}_6\texttt{ ==> leq(A,C).}$

for active constraint `leq(X,Y)`, traverses the list of `leq/2` constraints attached to the X variable, looking for constraints of the form `leq(_,X)`. This list may of course include constraints of the form `leq(W,f(X,Z))` and `leq(X,W)` which do not match. This is avoided in HAL by using an index on the second argument of the `leq/2` constraints which only finds those with X as the second argument. □

Abstractly speaking, an index maintains a mapping from some key K (built from solver terms) to a list of numbered CHR constraints "matching" that key, i.e., constraints containing the solver terms in K in the appropriate argument positions. The HAL compiler determines, for each occurrence, which indexes are required for supporting the efficient computation of its partners. A key point is that indexes implicitly *ask equality constraints*. The following example makes this clearer.

EXAMPLE 19. Consider the indexes required for some of the occurrences in the `leq` program in Figure 3.

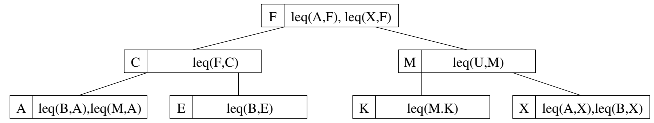

Figure 8: A tree of lists of CHR constraints indexed on the second argument

$\texttt{leq(A,B)}_3\texttt{, leq(B',A')}_2\texttt{ <=> A = A', B = B' | A = B.}$
$\texttt{leq(A,B)}_7\texttt{, leq(B',C)}_6\texttt{ ==> B = B' | leq(A,C).}$

For the second occurrence, with active constraint `leq(X,Y)` we are looking for a CHR constraint of the form `leq(Y,X)` to ensure the guard constraints $A = A', B = B'$ hold. Hence, we need an index on both arguments of `leq/2`. Similarly, for the third occurrence. For the sixth occurrence, we need to find CHR constraints of the form `leq(_,X)` to ensure that $B = B'$ holds. Thus, we need an index on the second argument. □

The HAL CHR compiler for CHRs over ground data suports several structures (such as trees and lists) as indexes, with tree indexes being used by default. We plan to support hash-based indexes in the future.

The core of the tree index code is an ordering $\preceq_u$ over solver terms according to the current state u of the solver. This order is used to traverse the tree. We write $x \equiv_u y$ when $x \preceq_u y \wedge y \preceq_u x$ and $x \prec_u y$ when $x \preceq_u y \wedge \neg(y \preceq_u x)$. The ordering must be *total* and is assumed to satisfy the following *soundness* property: if $x \preceq_u y \wedge y \preceq_u x$ then $u \models x = y$. Thus, the ordering answers whether equality constraints are *entailed* by the current store u.

For data that involves no solver variables, the precedence relation $\preceq_u$ does not depend on the solver state. As a result the relationship between two such terms in the ordering cannot change. However, when the value of the data is not fixed, the ordering relationship between two terms can change as the solver state evolves.

EXAMPLE 20. The usual ordering $\prec_u$ of two Herbrand terms (Prologs `@<`) is defined as the lexicographic ordering (with variables before functors) of the solved form of the terms involved. In the empty solver state $u_0 = \emptyset$ we have $X \prec_{u_0} Y$, but when $u_1 \equiv \{X = f(A)\}$ we have $Y \prec_{u_1} X$, but then adding another equation $u_2 \equiv \{X = f(A), Y = f(B)\}$ we have $X \prec_{u_2} Y$, and finally adding $A = B$, $u_3 \equiv \{X = f(A), Y = f(B), A = B\}$ we have that $X \equiv_{u_3} Y$. □

As illustrated above, during forward computation as the state u changes by adding new constraints to u', the order between solver terms can change in arbitrary ways (although we can always ensure that if $x \equiv_u y$ then also $x \equiv_{u'} y$). Therefore, a tree index based on comparison results which is correct at u, may become corrupted at u'.

EXAMPLE 21. Figure 8 shows a binary search tree for `leq/2` constraints indexed on the second argument based on the ordering $\prec_\emptyset$ of an empty store. When we add the constraint $C = M$ the ordering $\prec_{\{C=M\}}$ uses the solved form of the terms. Assuming M is replaced by C, then searching in the tree for entry with key K will not succeed. If C is replaced by M then the same applies to E. □

The simplest approach to repairing a tree index which was correct for state u and is corrupted for u' is as follows. We must first delete all entries where $\preceq_u$ and $\preceq_{u'}$ are not guaranteed to be the same, thus obtaining a correct tree. Then, the re-insertion can be made using the new ordering $\preceq_{u'}$. There is however a slight problem: the deletion of the corrupted entries has to be performed *before* u actually changes into u', so that these entries can be correctly located.

EXAMPLE 22. Consider the tree in Figure 8, when we add the constraint $C = M$, and assume the ordering $\preceq_{\{C=M\}}$ replaces M by C. If when trying to locate the corrupted entry M we use the *current* ordering $\preceq_{\{C=M\}}$, then at the root we will find $M \prec_{\{C=M\}} F$, go left and find $M \equiv_{\{C=M\}} C$. Unfortunately, we will have not discovered the M node, but the C node. In order to correctly locate the M node we need to use the previous ordering $\preceq_\emptyset$. □

This problem is solved in HAL by requiring the solver to support two things. The first, is the comparison predicate `compare(Result,X,Y)` which returns $=$, $<$ or $>$ when in the current solver state u respectively $X \equiv_u Y$, $X \prec_u Y$ and $Y \prec_u X$. The second, is a new wake condition `cc(X)` (*compare change*) which holds whenever a change in the solver state might cause the result of a comparison involving X to change (usually highly related to the wake list for `X == Y`). Importantly, the `cc(X)` condition fires and executes any delayed goals "just before" the change in solver state occurs.[6]

EXAMPLE 23. The HAL Herbrand solver supports the `cc(X)` condition. Thus, when using CHRs that extend the Herbrand solver we can setup delayed goals on this condition to delete the modified constraints. The following pseudocode inserts a `leq/2` constraint into two indexes, the first (`index1`) indexes on the second argument Y, while the second (`index2`) indexes on both arguments (X, Y):

```
insert_leq(X,Y,Id) :-
   (  cc(X) ==> index2_delete((X,Y),leq(X,Y,Id))
   || cc(Y) ==> index1_delete(Y,leq(X,Y,Id)),
                index2_delete((X,Y),leq(X,Y,Id)) ),
   index1_insert(Y,leq(X,Y,Id)),
   index2_insert((X,Y),leq(X,Y,Id)).
```

Returning to the problem of Example 22, when $C = M$ is added, just before the solver state (in this case the heap) is changed, the goal `index1_delete(M,leq(U,M,Id))` is executed deleting the node with key M. □

The implementation of the `cc` condition may be burdensome for the solver writer. For solvers other than Herbrand and those whose comparison relies on reference types, there is a simpler solution. This is because the only heap data that is destructively updated is cells for Herbrand variables and reference types. For solvers not using this kind of data we can use the universal `system_compare` predicate, which compares terms using their heap representation. This means we can miss potential partners since we only detect equivalence of two terms whose heap representation is identical. However, the implementation is sound since the indexes cannot be corrupted, and there is no need to support the `cc` wake condition.

[6] Attributed variables in most Prologs similarly must interrupt unification in order to perform computation just before the heap changes.

EXAMPLE 24. Variables in the `fdint` solver must have type `fdint` which is defined as

```
:- typedef fdint -> val(int) ; var(int).
```

i.e., they are either known to have a fixed `int`eger value from the start or they are initialised as an `int` numbered variable (which is a pointer into a global array of variable information). We can define `compare` so that it simply compares these representations. This definition is correct (i.e., if it returns true the ask constraint holds), but it is incomparable with the definition of `==`. Given a solver state where variable `var(27)` has the singleton domain $\{3\}$, and `var(42)` has domain $\{4, 6, 7\}$, then `compare(R,var(27),val(3))` returns `R = (<)` while `var(27) == val(3)` succeeds. Also, `compare(R,var(42),var(42))` returns `R = (=)` while `var(42) == var(42)` fails.

This means that `compare` does not depend on the solver state, so we do not need to update the index. However, we will miss partner constraints when we use the index. For example, when we look for variables equivalent to `var(27)` we will not find `val(3)`. □

We can do better if the solver can define `==` as mapping each argument to a "representation" which is also a legitimate solver term, and then checking these for identity. Because the CHR index structures are going to be revisited whenever the answer to the associated ask constraints might change, we can replace a solver term by its new representation.

EXAMPLE 25. For the `fdint` solver we can define the representation of a variable as itself, unless it is fixed (the solver predicate `val(X,XV)` succeeds returning in `XV` the value of `X`) in which case it is represented as a `val` term. The representation calculation is thus

```
rep(X,R) :- (val(X,XV) -> R = val(XV) ; R = X).
```

Let us now re-examine the previous example. When `var(27)` becomes fixed to 3, its representation changes from `var(27)` to `val(3)`, and the condition `fixed(var(27))` fires. Thus, we can attach code to this condition to delete the entries indexed by `var(27)` and reinsert them using `val(3)`. Note that we can do this after the solver state has changed, because the comparison results have not changed.

Pseudo-code illustrating this approach, assuming an index on the first two arguments of `min/3`, is given below. Whenever a variable in a key is fixed, its entries are deleted from the index, their representation is calculated, they are reinserted, and then re-executed.

```
insert_min(X,Y,Z,Id) :-
   (  fixed(X) ==> index_delete((X,Y),min(X,Y,Z,Id)),
                   rep(X,Xr), insert_min(Xr,Y,Z,Id),
                   min(Xr,Y,Z,Id)
   || fixed(Y) ==> index_delete((X,Y),min(X,Y,Z,Id)),
                   rep(Y,Yr), insert_min(X,Yr,Z,Id),
                   min(X,Yr,Z,Id) ),
   index_insert((X,Y),min(X,Y,Z,Id)).
```

□

In practice, code for checking occurrences of CHR constraints tries to delay the insertion of the constraint as long as possible, because often the active constraint is deleted. This avoids index manipulations for insertions and deletions, and can give considerable performance improvements. We can also gain these benefits in re-execution, by changing the re-execution code to delete the constraints from all its indexes and then recheck the appropriate occurrences, re-inserting it as late as possible.

Note that the issues we dealt with here are also mostly applicable to hash-based indexes over solver variables. Clearly, we must delete entries from the hash table before the result of the hash function changes for those items, hence the same issues arise.

7. EXPERIMENTS

The HAL CHR compiler has been extended to include `asks` declarations, to translate guard constraints to ask constraints, and to use wake conditions to set up minimal re-execution when a solver changes. It also automatically builds appropriate index structures for the joins required by CHR rules, but at present the code for the update of these index structures in the presence of changes in the solver is hand coded.

The code produced by the compiler is considerably more complicated than the example code used in the paper since we have omitted details related to typing and moding. The CHR compiler does not yet support the extension of solver types that include other solver types. To do so seems to require fairly ingenious use of overloading.

In this section we show the benefits of re-checking CHR rules using our optimized compilation scheme and the use of index structures over solver types using benchmarks for three different solvers. All timings are the average over 10 runs on a Dual Pentium II with 648M of RAM running under Linux RedHat 6.2 with kernel version 2.2.9 and are given in milliseconds. SICStus Prolog 3.8.6 is run under compact code (no fastcode for Linux). We compare to SICStus CHRs where possible just to illustrate that the HAL CHR implementation is mature and competitive.

The first set of benchmarks uses a simple Boolean solver which extends a Herbrand solver, and is similar to those available at the CHR web site [5]. This solver requires no indexes since it only simplifies individual Boolean constraints. The comparison uses five simple Boolean benchmarks (most from [2]): the first `pigeon(`n`,`m`)` places n pigeons in m pigeon holes (the 24-24 query succeeds, while 8-7 fails); `schur(`n`)` Schurs's lemma for n (see [2]) (the 13 query is the largest n that succeeds); `queens(`n`)` the Boolean version of this classic problem; `mycie(`n`,`m`)` which colors a 5-colorable graph (taken from [13]) with n nodes and m edges with 4 colours; and `fulladder(`n`)` which searches for a single faulty gate in a n bit adder (see e.g. [12] for the case of 1 bit). Table 1 gives the execution times in milliseconds for a *base* compilation where every rule for a CHR constraint is rechecked each time a variable in the CHR constraint is modified, and compilation with *spec*ialised re-execution. The *dyn* column gives the times for an equivalent Boolean solver hand coded using the dynamic scheduling features of the Herbrand solver directly. The *SICS* column is the equivalent CHR code in SICStus Prolog 3.8.6. The final row provides the geometric mean for *base* and its relative value for other columns. For this CHR solver, since there are no expensive joins to avoid in the re-execution, the solver provides modest speedups of around 20%. Surprisingly, the specialized code is competitive with the hand coded dynamically scheduled version, even improving it on occasion. This is because the

Prog	*base*	*spec*	*dyn*	*SICS*
`pigeon(8,7)`	1550	1289	1139	21828
`pigeon(24,24)`	617	612	577	2195
`schur(13)`	24	17	20	205
`schur(14)`	152	117	136	2747
`queens(18)`	11596	9165	10458	161858
`mycie(23,71)`	2885	2376	2151	38511
`fulladder(5)`	561	483	543	24236
Geom. mean	677	82%	85%	1300%

Table 1: Results for benchmarks using a CHR Boolean solver extending the Herbrand solver.

optimized CHR code can sometimes avoid setting up delayed goals, which the hand coded solver did not attempt to do. The HAL CHR code is obviously competitive with SICStus CHRs.

The next set of benchmarks uses the `min` solver of Figure 5 which extends a simple finite domain solver written directly in HAL. We use two program benchmarks, the first is an artificial example `min(`n`,`m`)` with a single constraint `min(X,Y,Z)` and m additional `min` constraints on other variables, where we progressively add the constraints

$$1 \leq X, 1 \leq Y, 1 \leq Z, X \leq n-1, Y \leq n-1, Z \leq n-1,$$
$$2 \leq X, 2 \leq Y, 2 \leq Z, X \leq n-2, Y \leq n-2, Z \leq n-2, ...$$

until only one value is left. The second example `bridge(`n`)` is the bridge scheduling problem of [14] where redundant `min` constraints are added to reduce the search. We use the (HAL version of) code in [12](page 277). We search for a solution which requires less than n days. When $n = 75$ there is no solution but the `min` constraints are more active. When $n = 112$ there is a solution.

We use four versions of the CHR code, the *base* version uses simple lists to store CHR constraints and rechecks each rule whenever a variables domain changes (as in Example 8), *spec* specializes the re-execution (as described in Example 12), *tree* uses tree indexes (implemented as in Example 25) and *both* does both. The results in Table 2 show that specialization is very important when there are no indexes available. Even when there are no partners to try to join with re-execution, specialization improves around 30%. When it can avoid expensive joins the benefit is unlimited. Adding tree indexes is clearly important for the artificial example when there are other min constraints, otherwise there is some overhead. Combining the optimizations the specialization is beneficial but less so. For the bridge example, tree indexes are not as helpful as in the artificial benchmark providing 10-20% improvement, and re-execution specialization adds an additional 5%. SICStus CHRs do not extend a finite domain solver, so we cannot compare with SICStus.

The next experiment uses the `leq` CHR program of Figure 1 which extends the Herbrand solver on two different benchmarks. The `leq(`n`,`m`)` benchmark builds a chain of variables of the form

$$X_0 \leq X_1 \leq ... \leq X_{n-1} \leq X_n$$

plus an additional m constraints of the form $X_i \leq X'$ (where X' represents a fresh variable), for each X_i in the chain, for a total of $n \times (m+1)$ `leq` constraints. Finally, it unifies $X_0 = X_n$ which gets the action going. The `scc` benchmark adds `leq` constraints representing the module dependency

Prog	*base*	*spec*	*tree*	*both*
`min(300,0)`	118	92	151	98
`min(300,500)`	6282	394	226	172
`min(300,1000)`	12841	1162	296	236
`min(600,0)`	489	360	590	358
`min(600,500)`	24485	689	682	428
`min(600,1000)`	48120	1450	744	504
`min(900,0)`	1081	804	1322	801
`min(900,500)`	54136	1153	1410	860
`min(900,1000)`	106521	2013	1498	944
`bridge(75)`	40116	31179	32644	30901
`bridge(112)`	78496	68668	71732	68777
Geom. mean	10067	14%	13%	9.1%

Table 2: Results for benchmarks using a CHR `min` solver extending a finite domain solver.

Prog	*base*	*spec*	*tree*	*both*	*SICS*
`leq(30,0)`	398	404	224	182	533
`leq(30,1)`	3006	2816	559	442	4397
`leq(30,2)`	8851	8189	916	725	11355
`leq(40,0)`	1233	1229	515	399	1369
`leq(40,1)`	9719	9154	1242	953	14031
`leq(40,2)`	29399	27390	2051	1612	38673
`leq(50,0)`	2946	2950	981	759	2937
`leq(50,1)`	24948	23616	2342	1781	36247
`leq(50,2)`	76565	71822	3806	2983	103811
`scc`	7422	6924	310	284	4892
Geom. mean	6595	96%	14%	11%	120%

Table 3: Results for benchmarks using a CHR `leq` solver extending a Herbrand solver.

graph for the HAL compiler. If there is a cycle, variables representing the modules are unified, thus calculating the transitive module dependencies and module cycles.

The results are shown in Table 3. For the artificial benchmark, we can see that re-execution specialization is not beneficial when there are no additional constraints, but improves around 7% when there are. The tree index is highly beneficial since the `leq` code involves many joins. Specialization is more effective with tree indexes, obtaining benefits between 20–25%. For the `scc` example specialization gives a 7% improvement on list indexes, and 9% when tree indexes are used. The attributed variable lookup mechanism of SICStus does not appear to be as efficient as tree indexes (although a direct comparison is not possible, due to the high number of other important differences).

8. CONCLUSION AND FURTHER WORK

In this paper we have shown how we can extend arbitrary solvers supporting dynamic scheduling using CHRs rules. We investigated how to compile the execution of CHR rules so that we re-execute the minimum number of rules on a change of solver state, and how to build and maintain indexes over the changing state of solver variables. Experiments show that specializing the re-execution is beneficial, and that the more expensive the joins in the CHR rules the more beneficial it is. The use of efficient indexes is vital to support efficient execution of CHRs, and hence we need to support such indexes over changing solver data.

9. REFERENCES

[1] S. Abdennadher. Operational semantics and confluence of constraint propagation rules. In Gert Smolka, editor, *Proceedings of the Third International Conference on Principles and Practice of Constraint Programming*, pages 252–266, 1997.

[2] P. Codognet and D. Diaz. Boolean constraint solving using `clp(FD)`. In *Procs. of ILPS'1993*, pages 525–539. MIT Press, 1993.

[3] 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.

[4] B. Demoen, M. García de la Banda, W. Harvey, K. Marriott, and P.J. Stuckey. An overview of HAL. In *Proceedings of the Fourth International Conference on Principles and Practices of Constraint Programming*, pages 174–188, 1999.

[5] T. Frühwirth. CHR home page. www.informatik.uni-muenchen.de/~fruehwir/chr/.

[6] T. Frühwirth. Theory and practice of constraint handling rules. *Journal of Logic Programming*, 37(1–3):95–138, 1998.

[7] 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, 2001.

[8] C. Holzbaur and T. Frühwirth. Compiling constraint handling rules into Prolog with attributed variables. In Gopalan Nadathur, editor, *Proceedings of the International Conference on Principles and Practice of Declarative Programming*, number 1702 in LNCS, pages 117–133. Springer-Verlag, 1999.

[9] C. Holzbaur and T. Frühwirth. Constraint handling rules, special issue. *Journal of Applied Artificial Intelligence*, 14(4), 2000.

[10] 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.

[11] JCK: Java constraint kit. http://www.pms.informatik.uni-muenchen.de/software/jack/index.html, 2002.

[12] K. Marriott and P.J. Stuckey. *Programming with Constraints: an Introduction.* MIT Press, 1998.

[13] M. Trick. mat.gsia.cmu.edu/COLOR/color.html.

[14] P. Van Hentenryck. *Constraint Satisfaction in Logic Programming.* MIT Press, 1989.