*Optimizing Compilation of Constraint Handling Rules in HAL**

CHRISTIAN HOLZBAUR

Dept. of Medical Cybernetics and Art. Intelligence, University of Vienna, Austria

MARIA GARCIA DE LA BANDA

School of Computer Science & Software Engineering, Monash University, Australia

PETER J. STUCKEY, GREGORY J. DUCK

Department of Computer Science & Software Engineering, University of Melbourne, Australia

Abstract

In this paper we discuss the optimizing compilation of Constraint Handling Rules (CHRs). CHRs are a multi-headed committed choice constraint language, commonly applied for writing incremental constraint solvers. CHRs are usually implemented as a language extension that compiles to the underlying language. In this paper we show how we can use different kinds of information in the compilation of CHRs in order to obtain access efficiency, and a better translation of the CHR rules into the underlying language, which in this case is HAL. The kinds of information used include the types, modes, determinism, functional dependencies and symmetries of the CHR constraints. We also show how to analyze CHR programs to determine this information about functional dependencies, symmetries and other kinds of information supporting optimizations.

1 Introduction

Constraint handling rules (Frühwirth, 1998) (CHRs) are a very flexible formalism for writing incremental constraint solvers and other reactive systems. In effect, the rules define transitions from one constraint set to an equivalent constraint set. Transitions serve to simplify constraints and detect satisfiability and unsatisfiability. CHRs have been used extensively (see e.g. (Holzbaur & Frühwirth, 2000)). Efficient implementations have been available for many years in the languages SICStus Prolog and Eclipse Prolog, and implementations for other languages are appearing such as Java (JCK, 2002) and HAL.

In this paper we discuss how to improve the compilation of CHRs by using additional information derived either from declarations provided by the user or from the analysis of the constraint handling rules themselves. The major improvements we discuss over previous work on CHR compilation (Holzbaur & Frühwirth, 2000) are:

---

* A preliminary version of this paper appeared under the title "Optimizing Compilation of Constraint Handling Rules" in *ICLP 2001*, Cyprus, November 2001 (Holzbaur *et al.*, 2001).

- general index structures which are specialized for the particular joins required in the CHR execution. Previous CHR compilation was restricted to two kinds of indexes: simple lists of constraints for given `Name/Arity` and lists indexed by the variables involved. For ground usage of CHRs this meant that only list indexes are used.
- continuation optimization, where we use matching information from rules earlier in the execution to avoid matching later rules.
- optimizations that take into account algebraic properties such as functional dependencies, symmetries and the set semantics of the constraints.

We illustrate the advantages of the various optimizations experimentally on a number of small example programs in the HAL implementation of CHRs. We also discuss how the extra information required by HAL in defining CHRs (that is, type, mode and determinism information) is used to improve the execution.

In part some of the motivation of this work revolves around a difference between CHRs in Prolog and in HAL. HAL is a typed language which does not (presently) support attributed variables. Prolog implementations of CHRs rely on the use of attributed variables to provide efficient indexing into the constraint store. Hence, we are critically interested in determining efficient index structures for storing constraints in the HAL implementation of CHRs. An important benefit of using specific index structures is that CHRs which are completely ground can still be efficiently indexed. This is not exploited in the current Prolog implementations. As some CHR solvers only use ground constraints this is an important issue.

The remainder of the paper is organized as follows. In the next section we give preliminary definitions, including the operational semantics of constraint handling rules. In Section 3 we go through the basic steps involved in compiling a set of constraint handling rules, and see how we can make use of properties such as functional dependencies and symmetry and set semantics in improving this basic compilation. In Section 4 we show how we can improve the the compilation of a set of CHRs by discovering properties of constraints by reasoning about the form of the rules defining them. In Section 5 we show how to infer the functional dependencies and symmetry information, used in Section 3, from a set of CHRs. In Section 6 we give our experimental results illustrating the advantages of the optimized compilation. Finally, in Section 7 we conclude.

2 Constraint Handling Rules and HAL

Constraint Handling Rules manipulate a global multiset of primitive constraints, using multiset rewrite rules which can take three forms

$$
\begin{aligned}
&\textit{simplification} \quad && [name@] \; c_1, \ldots, c_n \;\; \Longleftrightarrow \;\; g \mid d_1, \ldots, d_m \\
&\textit{propagation} \quad && [name@] \; c_1, \ldots, c_n \;\; \Longrightarrow \;\; g \mid d_1, \ldots, d_m \\
&\textit{simpagation} \quad && [name@] \; c_1, \ldots, c_l \setminus c_{l+1}, \ldots, c_n \;\; \Longleftrightarrow \;\; g \mid d_1, \ldots, d_m
\end{aligned}
$$

where *name* is an optional rule name, $c_1, \ldots, c_n$ are CHR constraints, g is a conjunction of constraints from the underlying language, and $d_1, \ldots, d_m$ is a conjunction

of CHR constraints and constraints of the underlying language. The guard part g is optional. If omitted, it is equivalent to $g \equiv true$.

The simplification rule states that given a constraint multiset $\{c'_1, \ldots, c'_n\}$ and substitution θ matching the multiset $\{c_1, \ldots, c_n\}$, i.e. $\{c'_1, \ldots, c'_n\} = \theta(\{c_1, \ldots, c_n\})$, where the execution of $\theta(g)$ succeeds, then we can replace $\{c'_1, \ldots, c'_n\}$ by multiset $\theta(\{d_1, \ldots, d_m\})$. The propagation rule states that, for a matching constraint multiset $\{c'_1, \ldots, c'_n\}$ where $\theta(g)$ succeeds, we should add $\theta(\{d_1, \ldots, d_m\})$. The simpagation rules states that, given a matching constraint multiset $\{c'_1, \ldots, c'_n\}$ where $\theta(g)$ succeeds, we can replace $\{c'_{l+1}, \ldots, c'_n\}$ by $\theta(\{d_1, \ldots, d_m\})$. A *CHR program* is a sequence of CHRs.

More formally the logical interpretation of the rules is as follows. Let $\bar{x}$ be the variables occurring in $\{c_1, \ldots, c_n\}$, and $\bar{y}$ (resp. $\bar{z}$) be the other variables occurring in the guard g (resp. rhs $d_1, \ldots, d_m$) of the rule. We assume no variables not in $\bar{x}$ appear in both the guard and the rhs.[1] The logical reading is

$$simplification \quad \forall \bar{x}(\exists \bar{y}\ g) \to (c_1 \wedge \cdots \wedge c_n \leftrightarrow (\exists \bar{z}\ d_1 \wedge \cdots \wedge d_m))$$
$$propagation \quad \forall \bar{x}(\exists \bar{y}\ g) \to (c_1 \wedge \cdots \wedge c_n \to (\exists \bar{z}\ d_1 \wedge \cdots \wedge d_m))$$
$$simpagation \quad \forall \bar{x}(\exists \bar{y}\ g) \to (c_1 \wedge \cdots \wedge c_n \leftrightarrow (\exists \bar{z}\ c_1 \wedge \cdots \wedge c_l \wedge d_1 \wedge \cdots \wedge d_m))$$

The operational semantics of CHRs exhaustively apply rules to the global multiset of constraints, being careful not to apply propagation rules twice on the same constraints (to avoid infinite propagation). For more details see e.g. (Abdennadher, 1997). Although CHRs have a logical reading (see e.g. (Frühwirth, 1998)), and programmers are encouraged to write confluent CHR programs, there are applications where a predictable order of rule applications is important. Hence, the textual order of rules in the program is used to resolve rule applicability conflicts in favor of earlier rules.

The operational semantics is a transition system on a triple $\langle s, h, t \rangle_v$ of a set of (numbered) CHR constraints s, a conjunction of Herbrand constraints h, and a set of transitions applied, as well as a sequence of variables v. The logical reading of $\langle s, h, t \rangle_v$ is as $\exists \bar{y}(s \wedge h)$ where y are the variables in the tuple not in v. Since the variable component v never changes we omit it for much of the presentation.

The transitions are defined as follows: Given a rule numbered a and a tuple $\langle s, h, t \rangle_v$

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

where $d_1, \ldots, d_k$ are CHR literals and $d_{k+1}, \ldots, d_m$ are Herbrand constraints, such that there are numbered literals $\{c'_{i_1}, \ldots, c'_{i_n}\} \subseteq s$ where $\models h \to \exists \bar{x} c_1 = c_{i_1} \wedge \cdots \wedge c_n = c_{i_n}$ and there is no entry $(i_1, \ldots, i_n, a)$ in t then the transition can be performed to give new state $\langle s \cup \{d_1, \ldots, d_k\}, h \wedge d_{k+1} \wedge \cdots \wedge d_m, t \cup \{(i_1, \ldots, i_n, a)\} \rangle_v$ where the new literals in the first component are given new id numbers.

The rule for simplification is simpler. Given a tuple $\langle s, h, t \rangle_v$ and a rule

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

---

[1] This allows us to more easily define the logical reading, we can always place a CHR in this form, by copying parts of the guard into the right hand side of the rule and renaming.

such that there are literals $\{c'_{i_1}, \ldots, c'_{i_n}\} \subseteq s$ where $\models h \rightarrow \exists \bar{x} c_1 = c_{i_1} \wedge \cdots \wedge c_n = c_{i_n}$ the resulting tuple is $\langle s \setminus \{c'_{i_1}, \ldots, c'_{i_n}\} \cup \{d_1, \ldots, d_k\}, h \wedge d_{k+1} \wedge \cdots \wedge d_m, t\rangle_v$.

In this paper we focus on the implementation of CHRs in a programming language, such as HAL (Demoen *et al.*, 1999), which requires programmers to provide type, mode and determinism information. A simple example of a HAL CHR program to compute the greatest common divisor of two positive numbers a and b (using the goal `gcd(a), gcd(b)`) is given below.

```
:- module gcd.                                              (L1)
:- import int.                                              (L2)
:- chr_constraint gcd/1.                                    (L3)
:- export pred gcd(int).                                    (L4)
:- mode gcd(in) is det.                                     (L5)
base @ gcd(0) <=> true.                                     (L6)
pair @ gcd(N) \ gcd(M) <=> M >= N | gcd(M-N).               (L7)
```

The first line $(L1)$ states that the file defines the module `gcd`. Line $(L2)$ imports the standard library module `int` which provides (ground) arithmetic and comparison predicates for the type `int`. Line $(L3)$ declares the predicate `gcd/1` to be implemented by CHRs. Line $(L4)$ exports the CHR constraint `gcd/1` which has one argument, an `int`. This is the *type* declaration for `gcd/1`. Line $(L5)$ is an example of a *mode of usage* declaration. The CHR constraint `gcd/1`'s first argument has mode `in` meaning that it will be fixed (ground) when called. The second part of the declaration "`is det`" is a determinism statement. It indicates that `gcd/1` always succeeds exactly once (for each separate call). For more details on types, modes and determinism see (Demoen *et al.*, 1999; Somogyi *et al.*, 1996).

Lines $(L6)$ and $(L7)$ are the 2 CHRs defining the `gcd/1` constraint. The first rule is a simplification rule. It states that a constraint of the form `gcd(0)` should be removed from the constraint store to ensure termination. The second rule is a simpagation rule. It states that given two different `gcd/1` constraints in the store, such that one `gcd(M)` has a greater argument than the other `gcd(N)` we should remove the larger (the one after the \), and add a new `gcd/1` constraint with argument `M-N`. Together these rules mimic Euclid's algorithm.

The requirement of the HAL compiler to always have correct mode information means that CHR constraints can only have declared modes that do not change the instantiation state of their arguments,[2] since the compiler will be unable to statically determine when rules fire. Hence for example legitimate modes are `in`, which means the argument is fixed at call time and return time, and `oo`, which means that the argument is initialized at call time, but nothing further is known, and similarly at return time. The same restriction applies to dynamically scheduled goals in HAL (see (Demoen *et al.*, 1999)).

---

[2] They may actually change the instantiation state but this cannot be made visible to the mode system.

3 Optimizing the basic compilation of CHRs

Essentially, the execution of CHRs is as follows. Every time a new constraint (the *active constraint*) is placed in the store, we search for a rule that can fire given this new constraint, i.e., a rule for which there is now a set of constraints that matches its left hand side. The first such rule (in the textual order they appear in the program) is fired.

Given this scheme, the bulk of the execution time for a CHR

$$c_1, \ldots, c_l[, \backslash]c_{l+1}, \ldots, c_n \quad \begin{array}{c} \Longleftrightarrow \\ \Longrightarrow \end{array} \quad g \mid d_1, \ldots, d_m$$

is spent in determining partner constraints $c'_1, \ldots, c'_{i-1}, c'_{i+1}, \ldots, c'_n$ for an active constraint c'_i to match the left hand side of the CHR. Hence, for each rule and each occurrence of a constraint, we are interested in generating efficient code for searching for partners that will cause the rule to fire. We will then link the code for each instance of a constraint together to form the entire program for the constraint. In this section, when applicable, we will show how different kinds of compile-time information can be used to improve the resulting code in the HAL version of CHRs.

3.1 Join Ordering

The left hand side of a rule together with the guard defines a multi-way join with selections (the guard) that could be processed in many possible ways, starting from the active constraint. This problem has been extensively addressed in the database literature. However, most of this work is not applicable since in the database context they assume the existence of information on cardinality of relations (number of stored constraints) and selectivity of various attributes. Since we are dealing with a programming language we have no access to such information, nor reasonable approximations. Another important difference is that, often, we are only looking for the first possible join partner, rather than all. In the SICStus CHR version, the calculation of partner constraints is performed in textual order and guards are evaluated once all partners have been identified. In HAL we determine a best join order and guard scheduling using, in particular, mode information.

Since we have no cardinality or selectivity information we will select a join ordering by using the number of unknown attributes in the join to estimate its cost. Functional dependencies are used to improve this estimate, by eliminating unknown attributes from consideration that are functionally defined by known attributes

Functional dependencies are represented as $p(\bar{x}) :: S \rightsquigarrow x$ where $S \cup \{x\} \subseteq \bar{x}$ meaning that for constraint p fixing all the variables in S means there is at most one solution to the variable x. The function fdclose($Fixed$,FDs) closes a set of fixed variables $Fixed$ under the finite set of functional dependencies FDs. fdclose($Fixed$,FDs) is the least set $F \supseteq Fixed$ such that for each $(p(\bar{x}) :: S \rightsquigarrow x) \in FDs$ such that $S \subseteq F$ then $x \in F$. Clearly the fixedpoint exists, since the operation is monotonic.

We assume an initial set $Fixed$ of known variables (which arises from the active constraint), together with the set of (as yet unprocessed) partner constraints and

guards. The algorithm measure shown in Figure 1, takes as inputs the set $Fixed$, the sequence $Partners$ of partner constraints in a particular order, the set FDs of functional dependencies and the set $Guards$ of guards, and returns the triple $(Measure, Goal, Lookups)$.

$Measure$ is an ordered pair representing the cost of the join for the particular order given by the n partner constraints in $Partners$. It is made up of the weighted sum $(n-1)w_1 + (n-2)w_2 + \cdots + 1w_{n-1}$ of the costs w_i for each individual join with a partner constraint. The weighting encourages the cheapest joins to be earliest.

The cost of a joining the ith partner constraint to pre-join expression (the join of the active constraint plus the first $(i-1)$ partners), w_i, is defined as the pair (u, f): u is the number of arguments in the new partner which are unfixed before the join; and f is the negative of the number of arguments which are fixed in the pre-join expression. The motivation for this costing is based on the worst case size of the join, assuming each argument ranges over a domain of the same size s. In this case the number of join partners (tuples) in the partner constraint for each set of values for the pre-join expression is s^u, and there are s^{m-f} tuples in the pre-join expression (where m is the total number of variables in the pre-join expression). The total number of tuples after the ith partner is joined are the thus s^{m-f+u}. The numbers hence represent the exponents of the join size, a kind of "degrees of freedom" measurement. The sum of the first components u gives the total size of the join. The role of the second component is to prefer orderings which keep the intermediate results smaller.

We also take into account the selectivity of the guards we can schedule directly after the new partner. This is achieved via the $selectivity(Guards)$ function which returns the sum of the selectivities of the guards $Guards$. The selectivity of a equational guard $X = Y$ is 1 provided X and Y are both fixed, otherwise the selectivity is 0. An equation with both X and Y fixed immediately eliminates one degree of freedom (reduces the number of tuples by $1/s$), hence the selectivity of 1. When one variable is not fixed, the guard does not remove any answers. If both X and Y are fixed then the selection For simplicity, the selectivity of other guards is considered to be 0.5 (as motivation, the constraint $X > Y$ where X and Y can be considered to remove 0.5 degrees of freedom). The role of selectivity is to encourage the early scheduling of guards which are likely to fail.

$Goal$ gives the ordering of partner constraints and guards (with guards scheduled as early as possible). Finally, $Lookups$ gives the queries. Queries will be made from partner constraints, where a variable name indicates a fixed value, and an underscore (_) indicates an unfixed value. For example, query p(X,_,X,Y,_) indicates a search for p/5 constraints with a given value in the first, third, and fourth argument positions, the values in the first and third position being the same.

The function schedule_guards(F,G) returns which guards in G can be scheduled given the fixed set of variables F. Here we see the usefulness of mode information which allows us to schedule guards as early as possible. For simplicity, we treat mode information in the form of two functions: $invars$ and $outvars$ which return the set of input and output arguments of a guard procedure. We also assume that each guard has exactly one mode (it is straightforward to extend the approach to

```
measure(Fixed,Partners,FDs,Guards)
   Lookups := ∅; score := (0,0); sum := (0,0)
   Goal := schedule_guards(Fixed, Guards)
   Guards := Guards \ Goal
   while true
      if Partners = ∅ return (score, Goal, Lookups)
      let Partners ≡ p(x̄), Partners1
      Partners := Partners1
      FDₚ := {p(x̄) :: fd ∈ FDs}
      Fixedₚ := fdclose(Fixed, FDₚ)
      f̄ := x̄ ∩ Fixedₚ
      Fixed := Fixed ∪ x̄
      GsEarly := schedule_guards(Fixed, Guards)
      cost := (max(|x̄ \ f̄| − selectivity(GsEarly), 0), −|f̄| − selectivity(GsEarly))
      score := score + sum + cost; sum := sum + cost
      Lookups := Lookups ∪ {p((xᵢ ∈ f̄ ? xᵢ : _) | xᵢ ∈ x̄)}
      Goal := Goal, p(x̄), GsEarly;
      Guards := Guards \ GsEarly
   endwhile
   return (score, Goal, Lookups)


schedule_guards(F,G)
   S := ∅
   repeat
      G₀ := G
      foreach g ∈ G
         if invars(g) ⊆ F
            S := S, g
            F := F ∪ outvars(g)
            G := Gs \ {g}
   until G₀ = G
   return S
```

Fig. 1. Algorithm for evaluating join ordering

multiple modes and more complex instantiations). The schedule_guards keeps adding guards to its output argument while they can be scheduled.

The function measure works as follows: beginning from an empty goal, we first schedule all possible guards. We then schedule each of the partner constraints $p(\bar{x})$ in *Partners* in the order given, by determining the number of fixed ($\bar{f}$) and unfixed ($\bar{x} \setminus \bar{f}$) variables in the partner, and the selectivity of any guards that can be scheduled immediately afterwards. With this we the calculate *cost* pair for the join which is added into the *score*. The *Goal* is updated to add the join $p(\bar{x})$ followed by the guards that can be scheduled after it. When all partner joins are calculated the function returns.

Example 1

Consider the compilation of the rule:

```
p(X,Y), q(Y,Z,T,U), flag, r(X,X,U) \ s(W) ==> W = U + 1, linear(Z) | p(Z,W).
```

for active constraint p(X,Y) and $Fixed = \{X, Y\}$. The individual costs calculated

for each join in the left-to-right partner order illustrated in the rule are $(2.5, -1.5)$, $(0, 0)$, $(0, -2)$, $(0, -1)$ giving a total cost of $(10, -11)$ together with goal

`q(Y,Z,T,U), W = U + 1, linear(Z), flag, r(X,X,U), s(W)`

and lookups `q(Y,_,_,_)`, `flag`, `r(X,X,U)`, `s(W)`. The best order has total cost $(4.5, -7.5)$ resulting in goal

`flag, r(X,X,U), W = U + 1, s(W), q(Y,Z,T,U), linear(Z)`

and lookups `flag`, `r(X,X,_)`, `s(W)`, `q(Y,_,_,U)`.

   For active constraint `q(Y,Z,T,U)`, the best order has total cost $(2, -8)$ resulting in goal

`W = U + 1, linear(Z), s(W), flag, p(X,Y), r(X,X,U)`

and lookups `s(W)`, `flag`, `p(_,Y)`, `r(X,X,U)`. $\square$

   For rules with large-left-hand sides where examining all permutations is too expensive we can instead greedily search for a permutation of the partners that is likely to be cost effective. The current HAL implementation uses this method. In practice, both methods usually find the best ordering because the left-hand-sides of CHRs are generally small.

3.2 Index Selection

Once join orderings have been selected, we must determine for each constraint a set of lookups of constraints of that form in the store. We then select an index or set of indexes for that constraint that will efficiently support the lookups required. Finally, we choose a data structure to implement each index. Mode information is crucial to the selection of index data structures. If the terms being indexed on are not ground, then we cannot use tree indexes since variable bindings will change the correct position of data.[3]

   The current SICStus Prolog CHR implementation uses only two index mechanisms: Constraints for a given `Functor/Arity` are grouped, and variables shared between heads in a rule *index* the constraint store because matching constraints must correspondingly share a (attributed) variable. In the HAL CHR version, we put extra emphasis on indexes for ground data:

   The first step in this process is *lookup reduction*. Given a set of lookups for constraint `p/k` we reduce the number of lookups by using information about properties of `p`/k:

- lookup generalization: rather than build specialized indexes for lookups that share variables we simply use more general indexes. Thus, we replace any lookup $p(v_1, \ldots, v_k)$ where v_i and v_j are the same variable by a lookup $p(v_1, \ldots, v_{j-1}, v'_j, v_{j+1}, \ldots, v_k)$ where v'_j is a new variable. Of course, we must add an extra guard $v_i = v_j$ for rules where we use generalized lookups. For example, the lookup `r(X,X,U)` can use the lookup for `r(X,XX,U)`, followed by the guard `X = XX`.

---

[3] Currently HAL only supports CHRs with fixed arguments (although these might be variables from another (non-Herbrand) solver).

- functional dependency reduction: we can use functional dependencies to reduce the requirement for indexes. We can replace any lookup $p(v_1, \ldots, v_k)$ where there is a functional dependency $p(x_1, \ldots, x_k) :: \{x_{i_1}, \ldots, x_{i_m}\} \rightsquigarrow x_j$ and $v_{i_1}, \ldots, v_{i_m}$ are fixed variables (i.e. not _) by the lookup $p(v_1, \ldots, v_{j-1}, \_, v_{j+1}, \ldots, v_k)$. For example, consider the constraint `bounds(X,L,U)` which stores the lower `L` and upper `U` bounds for a constrained integer variable `X`. Given functional dependency $bounds(X, L, U) :: X \rightsquigarrow L$, the lookup `bounds(X,L,_)` can be replaced by `bounds(X,_,_)`.
- symmetry reduction: if `p/k` is symmetric on arguments i and j we have two symmetric lookups $p(v_1, \ldots, v_i, \ldots, v_j, \ldots, v_k)$ and $p(v'_1, \ldots, v'_i, \ldots, v'_j, \ldots, v'_k)$ where $v_l = v'_l$ for $1 \leq l \leq k, l \neq i, l \neq j$ and $v_i = v'_j$ and $v_j = v'_i$ then remove one of the symmetric lookups. For example, if `eq/2` is symmetric the lookup `eq(_,Y)` can use the index for `eq(X,_)`.

We discuss how we generate functional dependency and symmetry information in Section 5. We can now choose the data structures for the indexes that support the remaining lookups.

Normally, the index will return an iterator which iterates through the multiset of constraints that match the lookup. Conceptually, each index thus returns a list iterator of constraints matching the lookup. We can use functional dependencies to determine when this multiset can have at most one element. This is the case for a lookup $p(v_1, \ldots, v_k)$ with fixed variables $v_{i_1}, \ldots, v_{i_m}$ such that fd-close$(\{x_{i_1}, \ldots, x_{i_m}\}, FDp) \supseteq \{x_1, \ldots, x_k\}$ where FDp are the functional dependencies for `p/k`, since in this case the functional dependencies ensure that for fixed $v_{i_1}, \ldots, v_{i_m}$ there can be at most one tuple in the constraint. For example, the lookup `bounds(X,_,_)` returns at most one constraint given the functional dependencies: $bounds(X, L, U) :: X \rightsquigarrow L$ and $bounds(X, L, U) :: X \rightsquigarrow U$.

Since, in general, we may need to store multiple copies of identical constraints (CHR rules accept multisets rather than sets of constraints) each constraint needs to be stored with a unique identifier, called the *constraint number*. Code for the constraint will generate a new identifier for each new active constraint. Constraints that cannot have multiple copies stored at once are said to have *set semantics* (see section 4.4). In this case constraint numbers are not strictly necessary.

Each index for $p(v_1, \ldots, v_k)$, where say the fixed variables are $v_{i_1}, \ldots, v_{i_m}$, needs to support the following operations:

```
:- pred p_index_init.
:- mode p_index_init is det.

:- pred p_index_insert(arg1, ..., argk, constraint_num).
:- mode p_index_insert(in, ..., in, in) is det.

:- pred p_index_delete(arg1, ..., argk, constraint_num).
:- mode p_index_delete(in, ..., in, in) is det.

:- pred p_index_init_iterator(argi1, ..., argim, iterator).
:- mode p_index_init_iterator(in, ..., in, out) is det.
```

for initializing a new index, inserting and deleting constraints from the index and returning an iterator over the index for a given lookup. Note that the constraint number is an important extra argument for index manipulation. In HAL indexes are stored in global variables, which are destructively updated for initialization, deletions and insertions. The compiler generates code for the predicates `p_insert_constraint` and `p_delete_constraint` which insert and delete the constraint `p` from each of the indexes in which it is involved.

The current implementation supports three kinds of index structures:

- A `yesno` global variable
- A balanced 234 tree
- An unsorted list (the default)

By far the simplest index structure is a `yesno` global variable, which can have two states: a `no` state (meaning nothing is currently stored) or a `yes(C)` state, where C is the only constraint currently in the store. The compiler will generate a `yesno` index structure whenever it detects that it is not possible for multiple[4] `p`/k constraints to exist in the store at once. This is the case whenever constraint `p`/k has set semantics (no identical copies) and has the functional dependencies $p(\bar{x}) :: \emptyset \rightsquigarrow x_i$ for $x_i \in \bar{x}$ (all copies must identical). An example is the constraint `gcd/1` from the `gcd` example program in Section 2. Here the rule

```
gcd(N) \ gcd(M) <=> M >= N | gcd(M-N).
```

ensures one of the two `gcd/2` constraints (one must be active) will be deleted. Therefore only one can ever actually be in the store at once, hence a `yesno` index structure may be used.

If constraint `p`/k has set semantics and functional dependencies of the form $p(x_1, \ldots, x_i, x_{i+1}, \ldots, x_k) :: \{x_1, \ldots, x_i\} \rightsquigarrow x_j$ for all $i < j \leq k$ then the compiler will generate a balanced 234 tree index structure. In this case the constraint `p`/k can be thought of as defining a function from the key $(x_1, \ldots, x_i)$ to a value $(x_{i+1}, \ldots, x_k)$. For example, the constraint `bounds(X,L,U)` from the program in Figure A 1 in Appendix A has the functional dependencies $bounds(X, L, U) :: X \rightsquigarrow L$ and $bounds(X, L, U) :: X \rightsquigarrow U$, hence the compiler builds a 234 tree index structure with `X` as the key, and the tuple `(L,U)` as the value. In addition, if the constraint `p`/k has set semantics, but has no functional dependency, then we can still use a tree index by treating the entire constraint as the key. For example, the constraint `X != Y` from the `interval` program in Appendix A has set semantics, thus we can use a tree structure with `(X,Y)` as the key, and the empty set $\emptyset$ as the value.

The big advantage of tree structures is $O(\log(n))$ lookups whenever the key is fixed, compared with $O(n)$ lookups for unsorted lists. Even if the key is only partially fixed there is still a potential for considerable benefit. Suppose that (X, Y) is the key, then all keys of the form $(X, \_)$ will group together in the tree index because of HAL's default ground term ordering. As a result, we can still do fast searches by pruning large sections of the tree that are not of interest. The same is not true for

---

[4] These constraints do not have to be identical.

keys of the form $(\_, Y)$ but we can sometimes use symmetric reduction to do the faster $(Y, \_)$ lookup instead.

Example 2
The CHR constraint `!=/2` defined by rules

```
neqsym @   X != Y ==> Y != X.
neqlower @ X != Y, bounds(X,VX,VX), bounds(Y,VX,UY) ==> bounds(Y,VX+1,UY).
nequpper @ X != Y, bounds(X,VX,VX), bounds(Y,LY,VX) ==> bounds(Y,LY,VX-1).
```

has lookups `!=(X,_)` and `!=(_,Y)`, and `!=/(X,Y)` and is known to be symmetric in its two arguments. We can remove the lookup `!=(_,Y)` in favor of the symmetric `!=(Y,_)`, and then use a single balanced tree index for `!=(X,Y)` to store `!=/2` constraints since this can also efficiently retrieve constraints of the form `!=(X,_)`. □

   The advantage of using a tree is lost whenever there is a lookup which is not a prefix of the index. These lookups can be implemented using universal search over the tree, but this is particularly bad, since we need to construct a tree iterator, which is currently implemented as a tree to list conversion with high overhead. For simplicity, we currently do not use tree indices when at least one lookup is not a prefix of the key. Fortunately universal searches against the direction of the functional dependency are relatively rare in practice, and in the future implementations of universal searches might do away with the need for iterators altogether.

   The third and final type of index structure is an unsorted list. The advantage of a list index is fast $O(1)$ insertions, but the disadvantages are slow $O(n)$ lookups and deletions. However, if a constraint p/k is never deleted, and is often involved in universal searches, then a list is a logical choice for the index structure.

3.3 Code generation for individual occurrences of active constraints

Once we have determined the join order for each rule and each active constraint, and the indexes available for each constraint, we are ready to generate code for each occurrence of the active constraint. Two kinds of searches for partners arise: A universal search iterates over all possible partners. This is required for propagation rules where the rule fires for each possible matching partner. An existential search looks for only the first possible set of matching partners. This is sufficient for simplification rules where the constraints found will be deleted.

   We can split the constraints appearing on the left-hand-side of any kind of rule into two sets: those that are deleted by the rule (*Remove*), and those that are not (*Keep*). The partner search uses universal search behavior, up to and including the first constraint in the join which appears in *Remove*. From then on the search is existential. If the constraint has a functional dependency that ensures that there can be only one matching solution, we can replace universal search by existential search.

   For each partner constraint we need to choose an available index for finding the matching partners. Since we have no selectivity or cardinality information, we simply choose the index with the largest intersection with the lookup.

```
bounds_4(X,L1,U1,CN1) :-
    (bounds_index_exists_iteration(X,_,L2,U2,CN2),
        CN1 != CN2 ->
        bounds_remove_constraint(X,L1,U1,CN1),
        bounds_remove_constraint(X,L2,U2,CN2),
        bounds(X,min(L1,L2),max(U1,U2)), %% RHS
        bounds_4_succ_cont(X,L1,U1,CN1)
    ;
        bounds_4_fail_cont(X,L1,U1,CN1) %% try next rule
    ).
```

Fig. 2. Existential search code for the fourth occurrence of a `bounds/3` constraint

Example 3

Consider the compilation of the 1st occurrence of the `bounds/3` constraint in the rule (the fourth occurrence overall in the program in Figure A 1)

```
intersect @ bounds(X,L1,U1),bounds(X,L2,U2) <=>
                        bounds(X,max(L1,L2),min(U1,U2)).
```

Since the active constraint is in *Remove* the entire search is existential. The compilation produces the code in Figure 2.

The predicate `bounds_index_exists_iteration` iterates non-deterministically through the `bounds/3` constraints in the store using the index on the first argument. In the last 4 arguments it returns the 3 arguments of `bounds/3` as well as a unique constraint number identifying the instance of the `bounds/3` constraint.[5] Note we check that the matching `bounds/3` constraint has a different constraint number than the active constraint `CN1 != CN2`. The predicate `bounds_remove_constraint` removes the `bounds/3` from the store. If the matching succeeds, then afterwards we call the success continuation `bounds_4_succ_cont` (which will later be replaced by `true`), otherwise we call the failure continuation `bounds_4_fail_cont`.

The compilation for first occurrence of a `bounds/3` constraint in the rule (the second occurrence overall)

```
redundant @ bounds(X,L1,U1) \ bounds(X,L2,U2) <=> L1 >= L2, U1 <= U2 | true.
```

requires universal search for partners since it is not deleted if the rule succeeds. The compilation produces the code shown in Figure 3.

The predicate `bounds_index_init_iterator`, returns an iterator of `bounds/3` constraints resulting from looking up the index. `bounds_iteration_last` and `bounds_iteration_next` respectively succeed if the iterator is finished and return values of the next `bounds/3` (and its constraint number) as well as the new iterator. After the rule has fired, the predicate `bounds_alive` checks that the active constraint has not been deleted as a consequence of executing the right-hand-side.

---

[5] The iterator need only return the last 2 arguments of `bounds/3` and the constraint number, since the first argument must be known, but the compilation is more straightforward if it always returns all arguments.

```
bounds_2(X,L1,U1,CN1) :-
    bounds_index_init_iterator(X,I0),
    bounds_2_forall_iterate(X,L1,U1,CN1,I0),
    bounds_2_cont(X,L1,U1,CN1).

bounds_2_forall_iterate(X,L1,U1,CN1,I0) :-
    bounds_iteration_last(I0),
    bounds_3(X,L1,U1,CN1).
bounds_2_forall_iterate(X,L1,U1,CN1,I0) :-
    bounds_iteration_next(I0,_,L2,U2,CN2,I1),
    (L1 >= L2, U1 <= U2, CN1 != CN2 -> %% Guard
        bounds_remove_constraint(X,L2,U2,CN1), %% remove matched constraint
        true %% RHS
    ;
        true %% rule did not apply
    ),
    (bounds_alive(CN1) ->
        bounds_2_forall_iterate(X,L1,U1,CN1,I1)
    ;
        true %% active has been deleted
    ).
```

Fig. 3. Universal search code for the second occurrence of a `bounds/3` constraint

If the active constraint is still alive, then we continue looking for more matchings. Note that for universal search we (presently) do not separate fail and success continuations. □

Example 4

Consider the compilation of the 3rd occurrence of a `gcd/1` constraint in the program in the introduction (the second occurrence in (*L7*)) which is to be removed. Since the active constraint is in *Remove* the entire search is existential. The compilation produces the code `gcd_3` shown in Figure 4. The predicate `gcd_index_exists_iteration` iterates non-deterministically through the `gcd/1` constraints in the store using the index (on no arguments). It returns the value of the `gcd/1` argument as well as its constraint number. Next, the guard is checked. Additionally, we check that the two `gcd/1` constraints are in fact different by comparing their constraint numbers (`CN1 != CN2`). If a partner is found, the active constraint is removed from the store, and the body is called. Afterwards, the success continuation for this occurrence is called. If no partner is found the failure continuation is called.

   The compilation for second occurrence of a `gcd/1` constraint (the first occurrence in (*L7*)) requires universal search for partners. The compilation produces the code `gcd_2` shown in Figure 4. The predicate `gcd_index_init_iterator`, returns an iterator of `gcd/1` constraints resulting from looking up the index. Calls to `gcd_iteration_last` and `gcd_iteration_next` succeed if the iterator is finished and return values of the last and next `gcd/1` constraint (and its constraint number) as well as the new iterator. □

```
gcd_3(M,CN1) :-                        gcd_2_forall_iterate(N,CN1,I0) :-
    (gcd_index_exists_iteration(N,CN2),    gcd_iteration_last(I0),
    M >= N, CN1 != CN2 -> %% guard        gcd_insert_constraint(N,CN1).
        gcd_delete_constraint(M,CN1),  gcd_2_forall_iterate(N,CN1,I0) :-
        gcd(M-N), %% RHS                   gcd_iteration_next(I0,M,CN2,I1),
        gcd_3_succ_cont(M,CN1)             (M >= N, CN1 != CN2 -> %% guard
    ;   gcd_3_fail_cont(M,CN1) ).              gcd_delete_constraint(M,CN2),
                                              gcd(M-N) %% RHS
gcd_2(N,CN1) :-                            ;
    gcd_index_init_iterator(I0),              true %% rule did not apply
    gcd_2_forall_iterate(N,CN1,I0),       ),
    gcd_2_cont(N,CN1).                     (gcd_alive(CN1) ->
                                              gcd_2_forall_iterate(N,CN1,I1)
                                          ;
                                              true
                                          ).
```

Fig. 4. Code for existential partner search and universal partner search.

3.4 Joining the code generated for each constraint occurrence

After generating the code for each individual occurrence, we must join it all together in one piece of code. The occurrences are ordered by textual occurrence except for simpagation rules where occurrences after the \ symbol are ordered earlier than those before the symbol (since they will then be deleted, thus reducing the number of constraints in the store). Let the order of occurrences be $o_1, \ldots, o_m$. The simplest way to join the individual rule code for a constraint p/k is as follows: Code for p/k creates a new constraint number and calls the first occurrence of code $p\_o_1/k+1$. The fail continuation for $p\_o_j/k+1$ is set to $p\_o_{j+1}/k+1$. The success continuation for $p\_o_j/k+1$ is also set to $p\_o_{j+1}/k+1$ unless the active constraint for this occurrence is in *Remove* in which case the success continuation is *true*, since the active constraint has been deleted.

Example 5
For the `gcd` program the order of the occurrences is $1, 3, 2$. The fail continuations simply reflect the order in which the occurrences are processed: `gcd_1` continues to `gcd_3` which continues to `gcd_2` which continues to `true`. Clearly, the success continuation for occurrences 1 and 3 of `gcd/1` are `true` since the active constraint is deleted. The continuation of `gcd_2` is `true` since it is last. The remaining code for `gcd/1` is given in Figure 5.[6] □

4 Improving CHR compilation

In the previous section we have examined the basics steps for compiling CHRs taking advantage of type, mode, functional dependency and symmetries information. In this section we examine other kinds of optimizations that can be performed by analysis of the CHRs.

---

[6] Note that later compiler passes remove the overhead of chain rules and empty rules.

```
gcd(N) :-
    new_constraint_number(CN1),      gcd_1_succ_cont(_,_).
    gcd_insert_constraint(N,CN1),    gcd_1_fail_cont(N,CN1) :- gcd_3(N,CN1).
    gcd_1(N,CN1).
gcd_1(N,CN1) :-                      gcd_3_succ_cont(_,_).
    (N = 0 -> %% Guard            gcd_3_fail_cont(N,CN1) :- gcd_2(N,CN1).
        gcd_delete_constraint(N,CN1),
        true, %% RHS                 gcd_2_cont(_,_).
        gcd_1_succ_cont(N,CN1)
    ;   gcd_1_fail_cont(N,CN1)).
```

Fig. 5. Initial code, code for first occurrence and continuation code for `gcd`

4.1 Continuation optimization

We can improve the simple strategy for joining the code generated for each occurrence of a constraint by noticing correspondences between rule matchings for various occurrences. Suppose we have two consecutive occurrences with active constraints, partner constraints and guards given by the triples $(p(\bar{x}), c, g)$ and $(p(\bar{y}), c', g')$ respectively. Suppose we can prove that $\models (\bar{x} = \bar{y} \wedge (\bar{\exists}_{\bar{y}} c' \wedge g')) \rightarrow \bar{\exists}_{\bar{x}} c \wedge g$ (where $\bar{\exists}_V F$ indicates the existential quantification of F for all its variables not in set V). Then, anytime the first occurrence fails to match the second occurrence will also fail to match, since the store has not changed meanwhile. Hence, the fail continuation for the first occurrence can skip over the second occurrence.

Example 6
Consider the following rules which manipulate `bounds(X,L,U)` constraints.

```
ne @ bounds(X,L,U) ==> U >= L.
red @ bounds(X,L1,U1) \ bounds(X,L2,U2) <=> L1 >= L2, U1 <= U2 | true.
int @ bounds(X,L1,U1), bounds(X,L2,U2) <=> bounds(X,max(L1,L2),min(U1,U2)).
```

For the 4th and 5th occurrences in rule `int` the implication

$$(X_4 = X_5 \wedge \exists L2_4, U2_4 bounds(X_4, L2_4, U2_4)) \rightarrow \exists L1_5, U1_5 bounds(X_5, L1_5, U1_5)$$

(where we use subscripts to indicate which is the active occurrence) holds. Hence, the 5th occurrence will never succeed if the 4th fails. Since if the 4th succeeds the active constraint is deleted, the 5th occurrence can be omitted entirely. □

We can similarly improve success continuations. If we can prove for two consecutive occurrences $(p(\bar{x}), c, g)$ and $(p(\bar{y}), c', g')$ that $\models \neg \bar{\exists}_{\emptyset}(\bar{x} = \bar{y} \wedge (\bar{\exists}_{\bar{x}} c \wedge g) \wedge (\bar{\exists}_{\bar{y}} c' \wedge g'))$ then if the $p(\bar{x})$ occurrence succeeds the $p(\bar{y})$ occurrence will not. Hence, the success continuation of $p(\bar{x})$ can skip the $p(\bar{y})$ occurrence. Again, we can use whatever form of reasoning we please to prove the unsatisfiability. Clearly, this is only of interest when the $p(\bar{x})$ occurrence does not delete the active constraint.

Example 7
Consider the two occurrences of `p/2` in the rules:

```
p(X,Y), q(Y,Y,X,T) ==> X >= Y | ...
r(A,B,C), p(C,D) ==> C < D | ...
```

The constraint $X = C \land Y = D \land (\exists T\ q(Y, Y, X, T) \land X \geq Y) \land (\exists A, B, C\ r(A, B, C) \land C < D)$ is clearly unsatisfiable and the success continuation of the first occurrence of `p/2` can skip the second. $\square$

Currently the HAL CHR compiler performs very simple fail continuation based on basic implication reasoning about identical constraints and `true`. Furthermore, because of some subtle complications arising from the implementation of universal searches, the current HAL CHR compiler restricts continuation optimization to existential searches. The difficulty stems from deciding if the head of the rule fires or not, which is information that this optimization relies upon. For the existential case there is no problem, since matching is already a mere `semidet` test. However a universal search may succeed multiple times, so some additional mechanism for recording the number of times a rule fires must be introduced. One possible solution is thread a counter through the code for the universal search, and count the number of times the search succeeds. If the counter is zero after the universal search code exists, then proceed with the fail continuation, otherwise proceed with the success continuation. This approach may be implemented in future versions of the compiler.

4.2 Late Storage

The first action in processing a new active constraint is to add it to the store, so that when it fires, the store has already been updated. In practice, this is inefficient since it may quite often be immediately removed. We can delay the addition of the active constraint until just before executing a right-hand-side that does not delete the active constraint, and can affect the store (i.e., may make use of the CHR constraints in the store).

Example 8
Consider the compilation of `gcd/1`. The first and third occurrences delete the active constraint. Thus, the new `gcd/1` constraint need not be stored before they are executed. It is only required to be stored just before the code for the second occurrence. The call to `gcd_insert_constraint` can be moved to the beginning of `gcd_2`, while the calls to `gcd_delete_constraint` in `gcd_1` and `gcd_3` can be removed. This simplifies `gcd/1` code considerably, as illustrated in Figure 6.
    $\square$

The current implementation infers this information by a simple pre-analysis. We can consider a rule that does not delete the active constraint as *rhs-affects-store* if its right-hand-side calls a CHR constraint, or a local predicate which calls CHR constraints (directly or indirectly), or (to be safe) an external predicate which is not a library predicate. In future compiler implementations, when CHR constraints are allowed to have non-ground arguments, we must also ensure no left-hand-side variables can ever be bound by the right-hand-side. This is because the CHR execution semantics dictate that whenever the instantiation state of a constraint changes, we must immediately run that constraint again as the active. However since the current implementation only supports ground CHR constraints, this issue is not yet relevant.

```
gcd(N) :-
    new_constraint_number(CN1),
    gcd_1(N,CN1).
gcd_1(N,CN1) :-
    (N = 0 -> %% Guard
        true, %% RHS
        true %% success continuation
    ;
        gcd_3(N,CN1) %% fail continuation
    ).
gcd_3(M,CN1) :-
    (gcd_index_exists_iteration(N,CN2),
    M >= N, CN1 != CN2 -> %% guard
        gcd_delete_constraint(M,CN1),
        gcd(M-N), %% RHS
        true %% success continuation
    ;
        gcd_2(M,CN1) %% fail continuation
    ).
gcd_2(N,CN1) :-
    gcd_index_init_iterator(I0),
    gcd_2_forall_iterate(N,CN1,I0).
gcd_2_forall_iterate(N,CN1,I0) :-
    gcd_iteration_last(I0),
    gcd_insert_constraint(N,CN1).
gcd_2_forall_iterate(N,CN1,I0) :-
    gcd_iteration_next(I0,M,CN2,I1),
    (M >= N, CN1 != CN2 -> %% guard
        gcd_delete_constraint(M,CN2),
        gcd_insert_constraint(N,CN1), %% late insert
        gcd(M-N) %% RHS
    ;
        true %% rule did not apply
    ),
    (gcd_alive(CN1) ->
        gcd_2_forall_iterate(N,CN1,I1)
    ;
        true
    ).
```

Fig. 6. Simplified code for `gcd/2` with late storage

Good late storage analysis is very important because most of the other analysis listed in this paper depends on it. Analysis for detecting set semantics, functional dependencies, never stored constraints and symmetry all rely on late storage information.

4.3 Never Stored

A rule of the form

$$c \iff d_1, \ldots, d_m$$

where c is a single constraint, always eliminates constraints of form c from the store. In particular if c is the most general form of the constraint $p(x_1, \ldots, x_k)$,[7] and p/k does not need to be stored because of earlier occurrences of p/k in rhs-affects-store rules, then we don't need to store this constraint at all. The advantage of never-stored information is that any rules involving p/k will only wake up when c is the active constraint. The current implementation searches for instances of never-stored rules and uses this information to avoid unnecessary joins, and to avoid building redundant index structures.

Example 9
Consider a `fixed/1` constraint which succeeds if its argument is a variable with equal lower and upper bounds, defined by the rules:

```
bounds(X,V,V) \ fixed(X) <=> true.
fixed(X) <=> fail.
```

Both rules delete the active `fixed/1` constraint. Thus, there will never be a `fixed/1` constraint in the store and hence an active `bounds/3` constraint will never match the rule. Thus, the occurrence of `bounds/3` in this rule will not be considered when compiling `bounds/3`. □

4.4 Set semantics

Although CHRs use a multiset semantics, often the constraints defined by CHRs have a set semantics. The current implementation detects two different forms of set semantics. Either the program rules ensure duplicate copies of constraints are always deleted or duplicate copies will not affect the behaviour of the program. The distinction between the two forms affects how the compiler takes advantage of this information, but in both cases set semantics allows us to choose more efficient index structures.

A constraint p/k has *set semantics* if there is a rule which explicitly removes duplicates of constraints. That is, if there exists a rule of the form

$$p(x_1, \ldots, x_k) \; [, \backslash] \; p(y_1, \ldots, y_k) \iff g \mid d_1, \ldots, d_m$$

such that $\models x_1 = y_1 \land \cdots x_k = y_k \to \bar{\exists}_{\bar{x} \cup \bar{y}} g$ which occurs before any rules require p/k to be stored.

Example 10
The rule

```
bounds(X,L1,U1) \ bounds(X,L2,U2) <=> L1 >= L2, U2 >= U1 | true.
```

ensures that any new active `bounds/3` constraint identical to one already in the store will be deleted (it also deletes other redundant bounds information). Since it occurs before any rules requiring `bounds/3` to be stored the constraint has set semantics. □

---

[7] All arguments are pair-wise different variables.

A constraint also has set semantics if all rules in which it appears behave the same even if duplicates are present. This is a very common case since CHRs are used to build constraint solvers which (by definition) should treat constraint multisets as sets. Thus, a constraint p/k also has *set* semantics if

1. there are no rules which can match two identical copies of p/k
2. there are no rules that delete a constraint p/k without deleting all identical copies. An exception is when the right-hand-side of such a rule always fails.
3. there are no rules with occurrences of p/k that can generate constraints (on the right-hand-side) which do not have set semantics.

The current implementation uses a simple fixpoint analysis which can detect such constraints starting from the assumption that all constraints have set semantics. In each iteration a constraint which violates one of the rules above is deleted from the candidate set of those with set semantics. The iterations proceed until a fixpoint is reached.

For constraints p/k having this form we can safely add a rule of the form

$$p(x_1, \ldots, x_k) \setminus p(x_1, \ldots, x_k) \iff true.$$

This will avoid redundant work when duplicate constraints are added. We can also modify the index structures for this constraint to avoid the necessity of storing duplicates.

Example 11
Consider a constraint `eq/2` (for equality) defined by the CHR

```
eq(X,Y),bounds(X,LX,UX),bounds(Y,LY,UY) ==> bounds(Y,LX,UX),bounds(X,LY,UY).
```

Then, since `bounds/3` has set semantics, `eq/2` also has set semantics. If we add the additional rule

```
eq(X,Y), X != Y <=> fail.
```

then `eq/2` still has set semantics. Even though the additional rule might delete one copy only of the constraint `eq/2`, it does not matter because the rule leads to failure. □

Example 12
Adding the additional rule which deletes identical copies of constraints can improve the termination of the program. Consider the following rules which define symmetry for `!=` constraints

```
neqset @ X != Y \ X != Y <=> true.
neqsym @ X != Y ==> Y != X.
```

If we delete the rule `neqset` then rule `neqsym` is an infinite loop for any new `!=/2` active constraint. However if `!=/2` implicitly has set semantics, then we will automatically add the rule `neqset`, hence the program becomes terminating. □

5 Determining Functional Dependencies and Symmetries

In previous sections we have either explained how to determine the information used for an optimization (as in the case of rules which are rhs-affects-store) or assumed it was given by the user or inferred by the compiler in the usual way (as in type, mode and determinism). The only two exceptions (functional dependencies and symmetries) were delayed in order not to clutter the explanation of CHR compilation. The following two sections examine how to determine these two properties.

5.1 Functional Dependencies

Functional dependencies occur frequently in CHRs since we encode functions using relations. Suppose p/k need not be stored before occurrences in a rule of the form

$$p(x_1, \ldots, x_l, y_{l+1}, \ldots, y_k)[, \backslash] p(x_1, \ldots, x_l, z_{l+1}, \ldots, z_k) \iff d_1, \ldots, d_m \qquad (1)$$

where $x_i, 1 \leq i \leq l$ and $y_i, z_i, l+1 \leq i \leq k$ are distinct variables. This corresponds to the functional dependencies $p(x_1, \ldots, x_k) :: (x_1, \ldots, x_l) \rightsquigarrow x_i, l+1 \leq i \leq k$. For example, the rule `int` of Example 6 illustrates the functional dependencies $bounds(X, L, U) :: X \rightsquigarrow L$ and $bounds(X, L, U) :: X \rightsquigarrow U$. In addition, rule (1) deletes identical copies ensuring p/k has set semantics. Therefore there is at most one constraint in the store of the form $p(x_1, \ldots, x_l, \_, \ldots, \_)$ at any time. Likewise, any constraint p/k that has a rule which deletes identical copies of constraints can also be thought of as having the functional dependency $p(x_1, \ldots, x_k) :: (x_1, \ldots, x_k) \rightsquigarrow \emptyset$.

Another common way functional dependencies are expressed in CHRs is by rules of the form

$$p(x_1, \ldots, x_l, y_{l+1}, \ldots, y_k), p(x_1, \ldots, x_l, z_{l+1}, \ldots, z_k) \implies y_{l+1} = z_{l+1}, \ldots y_k = z_k$$

This leads to the same functional dependency as before, however it does not lead to set semantics behavior.

We can detect more functional dependencies if we consider multiple rules of the same kind. For example, the rules

$$p(x_1, \ldots, x_l, y_{l+1}, \ldots, y_k)[, \backslash] p(x_1, \ldots, x_l, z_{l+1}, \ldots, z_k) \iff g_1 | d_1, \ldots, d_m$$
$$p(x_1, \ldots, x_l, y'_{l+1}, \ldots, y'_k)[, \backslash] p(x_1, \ldots, x_l, z'_{l+1}, \ldots, z'_k) \iff g_2 | d'_1, \ldots, d'_{m'}$$

also lead to functional dependencies if $\models (\bar{y} = \bar{y}' \wedge \bar{z} = \bar{z}' \rightarrow (g_1 \vee g_2)$ is provable. However because of the difficulty in solving disjunctions, the current analysis is limited to the case where g_1 and g_2 are primitive integer or real constraints (not conjunctions of other constraints).

Example 13
The second rule for `gcd/1` written twice illustrates the functional dependency $\mathbf{gcd}(N) :: \emptyset \rightsquigarrow N$ since $N = M' \wedge M = N' \rightarrow (M \geq N \vee M' \geq N')$ holds:

```
gcd(N) \ gcd(M) <=> M >= N | gcd(M - N).
gcd(N') \ gcd(M') <=> M' >= N' | gcd(M' - N').
```

Making use of this functional dependency for `gcd/1` we can use a single global

`yesno` integer value (`$Gcd`) to store the (at most one) `gcd/1` constraint, we can replace the forall iteration by exists iteration, and remove the constraint numbers entirely. The resulting code (after unfolding) is

```
gcd(X) :-
    (X = 0 -> true              %% occ 1: guard -> rhs
    ; (yes(N) = $Gcd, X >= N    %% occ 3: gcd_index_exists_iteration, guard
        gcd(X-N)                %% occ 3: rhs
    ; (yes(M) = $Gcd, M >= X    %% occ 2: gcd_forall_iterate, guard
        $Gcd := yes(X),         %% occ 2: gcd_insert_constraint
        gcd(M-X)                %% occ 2: rhs
    ; $Gcd := yes(X)))).        %% late insert
```

□

5.2 Symmetry

Symmetry also occurs reasonably often in CHRs. There are multiple ways of detecting symmetries. A rule of the form

$$p(x_1, x_2, \ldots, x_k) \implies p(x_2, x_1, \ldots, x_k)$$

that occurs before any rule that requires p/k to be inserted induces a symmetry for constraint $p(x_1, \ldots, x_k)$ on x_1 and x_2, providing that no rule eliminates $p(x_1, x_2, \ldots, x_k)$ and not $p(x_2, x_1, \ldots, x_k)$.

Example 14
Consider a `!=/2` constraint defined by the rules:

```
neqset @ X != Y \ X != Y <=> true.
neqsym @ X != Y ==> Y != X.
neqlower @ X != Y, bounds(X,VX,VX), bounds(Y,VX,UY) ==> bounds(Y,VX+1,UY).
nequpper @ X != Y, bounds(X,VX,VX), bounds(Y,LY,VX) ==> bounds(Y,LY,VX-1).
```

the rule `neqsym @ X != Y => Y != X` illustrates the symmetry of `!=/2` w.r.t. X and Y, since in addition no rule deletes a (non-duplicate) `!=/2` constraint. □

A constraint may be symmetric without a specific symmetry adding rule. The general case is complicated and, for brevity, we simply give examples.

Example 15
The rule in Example 11 and its rewriting with $\{X \mapsto Y, Y \mapsto X\}$ are logically equivalent (they are variants illustrated by the reordering of the rule).

```
eq(X,Y),bounds(X,LX,UX),bounds(Y,LY,UY) ==> bounds(Y,LX,UX),bounds(X,LY,UY).
eq(Y,X),bounds(Y,LY,UY),bounds(X,LX,UX) ==> bounds(X,LY,UY),bounds(Y,LX,UX).
```

Hence, since this is the only rule for `eq/2`, the `eq/2` constraint is symmetric. □

Example 16
The following rules remove redundant inequalities:

```
eq(X,Y) \ X <= Y <=> true.
eq(Y,X) \ X <= Y <=> true.
```

They are symmetric for $eq(x,y)$ on x and y. $\square$

If every rule containing constraint p/k is symmetric on x_1 and x_2 with another rule then, the constraint is symmetric on x_1 and x_2. Hence $eq(x,y)$ is symmetric in x and y.

Note that we can take into account symmetries in other constraints when proving symmetry of rules. Hence for example the additional rule

```
X != Y, eq(X,Y) ==> fail.
```

is symmetric for $eq(x,y)$ on x and y because of the symmetry of `!=/2`.

Note that we can find more symmetry by starting from the assumption that every constraint is symmetric on all arguments and iteratively disproving this.

6 Experimental Results

The initial version of the HAL CHR compiler (reported in (Holzbaur *et al.*, 2001)) realized only some of the optimizations discussed herein, continuation optimization, simplistic join ordering and simplistic late storage. The current version fully implements most of the analysis and optimizations discussed in this paper, including

- early guard scheduling and join ordering;
- the discovery of functional dependencies, set semantics and symmetries (only the first case of symmetry is detected);
- late storage; and
- the building of specialized indexes which rely on this information.

Analysis is performed in an independent compilation phase, followed by an optimized code generation phase which builds the index structures amongst other things. Further improvements in the analysis phase are possible, such as better discovery of symmetries (implied from rules) and improved continuation optimization. Further improvements are also possible in the optimized code generation phase, such as generating multiple indexes for every lookup (currently, the compiler generates one index per constraint).

The cost of the CHR analysis is small compared with the cost of other tasks performed by the HAL compiler, such as type and mode analysis. The discovery of functional dependencies, set semantics and symmetries is generally very cheap since only a linear pass over the program is required. Late storage is slightly more expensive since a call graph must be constructed. Join ordering is potentially expensive for rules with a large number of partners in the head, however the current implementation uses a greedy algorithm which is generally much faster.

The currently implementation is a prototype designed to demonstrate how an optimizing CHR compiler will eventually be realized. To test the analysis and optimizations implemented by the prototype we compare the performance on 3 small programs:

- `gcd` as described in the paper, where the query (a,b) is `gcd(a),gcd(b)`.

Table 1. *Summary of the information extracted by the analysis phase of CHR compilation*

Program	Constraint	FD	Set	Sym
gcd	gcd(X)	$\emptyset \rightsquigarrow X$	yes	—
interval	bounds(X,L,U)	$\{X\} \rightsquigarrow L, U$	yes	—
interval	eq(X,Y)	—	yes	—
interval	geq(X,Y)	—	yes	—
interval	X != Y	—	yes	$\{X, Y\}$
interval	plus(X,Y,Z)	—	yes	—
dfa	line(X,Y)	—	yes	$\{X, Y\}$

- `interval`: a simple bounds propagation solver executing N-queens; where the query (a, b) is for a queens with each constraint added b times (usually 1, just here to illustrate the possible benefits from set semantics). The full code for the bounds propagation solver (module `interval`) can be found in in Appendix A.
- `dfa`: a visual parser for deterministic finite automatas (DFAs) building the DFA from individual graphics elements, e.g. circles, lines and text boxes. The constraints are all ground, and the compilation involves a single (indexable) lookup $line(\_, Y)$, has a single symmetry $line(X, Y) = line(Y, X)$ and no constraints (except `line/2`) have set semantics. In this program the rules are large multi-ways joins, e.g., the rule to detect an arrow from one state to another is:

```
circle(C1,R1), circle(C2,R2) \
line(P1,P2), line(P2,P1), line(P3,P2), line(P2,P3),
line(P4,P2), line(P2,P4), text(P5,T) <=>
    point_on_circle(P1,C1,R1), point_on_circle(P2,C2,R2),
    midpoint(P1,P2,P12), near(P12,P5) | arrow(P1,P2,T).
```

Notice that the rule is careful to delete symmetric copies of the constraint `line/2`. The query a finds a (constant) small DFA (of 10 elements) in a large set of a redundant arrows (each consisting of three lines and a text box).

A summary of the results from the analysis phase are shown in Figure 1. For the `gcd` program, the analysis infers[8] the functional dependency $\emptyset \rightsquigarrow X$ and set semantics from the rule

```
gcd(N) \ gcd(M) <=> M >= N | gcd(M-N).
```

---

[8] See example 13 in Section 5.1

Table 2. *Execution times (ms) for various optimized versions of the* `gcd` *program*

Benchmark	Query	Orig	+yesno	+det	Hand
gcd	(5000000,3)	1111	976	402	50
gcd	(10000000,3)	2314	2032	803	93
gcd	(50000000,3)	12412	11093	5095	475
gcd	(100000000,3)	24891	22240	10270	961

The compiler uses this information to build a `yesno` index structure, since the functional dependency combined with set semantics implies that only one `gcd(X)` constraint can ever be in the store at one time.

The next program, `interval`, is the most fruitful in terms of information discovered. The rules

```
bounds(X,L1,U1) \ bounds(X,L2,U2) <=> L1 >= L2, U2 >= U1 | true.
bounds(X,L1,U1), bounds(X,L2,U2) <=> bounds(X,max(L1,L2),min(U1,U2)).
```

leads to the the discovery of the set semantics of `bounds/3` and the functional dependencies $bounds(X, L, U) : X \rightsquigarrow L$ and $bounds(X, L, U) : X \rightsquigarrow U$. Therefore only one copy of the constraint `bounds(X,_,_)` can ever be in the store at one time. The resulting structure for `bounds/3` is a balanced 234 tree with (X) as the key and (L, U) as the value.

All of the other `interval` constraints at least have set semantics. Set semantics are inferred for constraints that behave the same even if multiple copies are stored. For example, the only rule involving the constraint `eq/2` is

```
equals @ eq(X,Y), bounds(X,LX,UX), bounds(Y,LY,UY) ==>
                       bounds(Y,LX,UX), bounds(X,LY,UY).
```

Since `bounds/3` has set semantics, then so has `eq/2`. The compiler uses this information to eliminate active `eq/2` constraints that already occur in the store. Thus redundant work is avoided, and potentially the size of indexes is reduced. In addition symmetry on the constraint `!=/2` is detected because of the symmetric rule

```
neqsym @ X != Y ==> Y != X.
```

Because of the benefit with symmetric reduction, the compiler will choose a balanced 234 tree index for `!=/2` with (X, Y) as the key.

Finally the `dfa` program turns out to be the least interesting in terms of information discovered. The only constraint with any useful attributes is `line/2`, which is symmetric and has set semantics. Again the compiler generates a balanced 234 tree index (for the same reasons as `!=/2` in the `interval` program). All other constraints use the default unsorted list index structure. However, because of the large size of the heads of rules in the `dfa` program, the most important optimization is join ordering and early guard scheduling.

The results `gcd`, `interval` and `dfa` are shown in Table 2, Table 3 and Table 4 respectively. All timings are the average over 20 runs on a 1200MHz AMD Athlon

Table 3. *Execution times (ms) for various optimized versions of the* `interval` *program*

Benchmark	Query	Orig	+tree	+det	+sym	+eq
`interval`	(12,1)	389	138	126	67	68
`interval`	(15,1)	1312	382	355	172	169
`interval`	(20,1)	6077	1602	1535	693	677
`interval`	(30,1)	73158	11728	12537	4943	4916
`interval`	(12,2)	556	184	167	107	69
`interval`	(15,2)	1824	532	471	283	167
`interval`	(20,2)	8658	2148	1984	1135	669
`interval`	(30,2)	110224	21950	18799	8522	5071

Table 4. *Execution times (ms) for various optimized versions of the* `dfa` *program*

Benchmark	Query	Prolog	+join	+treesym	+det
`dfa`	20	4987	30	20	19
`dfa`	50	69070	164	87	86
`dfa`	100	532278	612	271	267
`dfa`	200	too long	2804	1525	1536
`dfa`	400	too long	13058	7401	7370

Processor with 1Gb of RAM running under Debian GNU Linux 3.0 with kernel version 2.2.19, and are given in milliseconds. Any test taking more than 600000ms (10 minutes) is marked as "too long".

For `gcd` we first give times for the original output of the compiler *Orig* (uses a list index). In the version *+yesno* the list storage of constraints is replaced by a *+yesno* structure (using the functional dependency and set semantics). We can see a modest improvement here by just avoiding some overhead. Note that in *Orig* the list index for `gcd/2` never grows more than one item in length anyway, so we do not expect a significant improvement by replacing a singleton list with a `yesno` structure. In *+det* the determinism declarations of the compiled CHR code is altered to take into account the functional dependency. This is a low level optimization in which previously "nondeterministic" lookups can be declared `semidet` (can succeed at most once). Without this optimization they are declared `cc_nondet` which means although they may succeed more than once we are interested only in the first solution. This produces faster executable code since deterministic (including `semidet`) code can be compiled in a simpler way than nondeterministic code. Finally *Hand* uses the hand optimized implementation of `gcd`/1 shown in Example 13. Here we see a considerable improvement purely from removing all of the overhead generated by

the compiler (such as constraint numbers). We expect that future implementations of the compiler will be able to remove most of this excess overhead.

The second experiment we show is `interval` in Table 3. The original code $Orig$ uses list indexes for all constraints, version $+tree$ is where the list index on `bounds/3` has been replaced by a 234 tree index (using the functional dependency), $+det$ where some `cc_nondet` searches are correctly declared `semidet`, $+sym$ where the list index on `!=/2` has been replaced by a 234 tree index (because we can take advantage of symmetric reduction), and $+eq$ where identical copies of set semantic constraints are deleted. Here we can see a significant improvement when the list index for `bounds/3` is replaced by a 234 tree index. This is not surprising, since we are replacing $O(n)$ lookups (for lists) with $O(\log(n))$ lookups (for trees). Next the $+det$ optimization provides a slight improvement in most cases. However for some unknown reason the test $(30, 1)$ actually becomes slightly worse. Next the $+sym$ lets us take advantage of symmetric reduction, which means we can choose a 234 tree index for the constraint `!=/2`. Again this provides a significant improvement. Finally the $+eq$ optimization deletes identical copies of constraints before they run as the active constraint. The handling of set semantics is of considerable benefit when duplicate constraints are actually added, and doesn't add significant overhead when there are no duplicate constraints, hence it seems worthwhile.

The final example is the `dfa` program in Table 4. The code $Prolog$ has the default join ordering and guard scheduling used by existing Prolog implementations of CHR compilers. Recall that this means guards are tested strictly after the join operation, hence the dreadful performance on a program with large rules, such as the `dfa` example. For the previous examples the default join ordering and the best join ordering coincide. Enabling join ordering and early guard scheduling ($+join$) produces a massive improvement in running time (`dfa` 100 is nearly 2000 times better). This highlights the importance of this optimization. Next $+treesym$ turns on 234 tree indexes and symmetric lookup reduction for the `line/2` constraint (without the symmetry lookup reduction, the compiler will not choose to use a 234 index because of a lookup $line(\_, Y)$). Once again we get a significant improvement. Finally in this case the $+det$ optimization seems to produce a very slight improvement, if at all.

Finally we remark that it is easy to optimize a very poor base implementation of CHRs. The HAL base implementation is highly efficient. The execution times of the $Orig$ or $Prolog$ columns are about an order of magnitude faster than CHRs in SICStus Prolog. See (Holzbaur *et al.*, 2001) for more detail.

7 Conclusion and Future Work

The core of compiling CHRs is a multi-way join compilation. But, unlike the usual database case, we have no information on the cardinality of relations and index selectivity. We show how to use type and mode information to compile efficient joins, and automatically utilize appropriate indexes for supporting the joins. We show how set semantics, functional dependencies and symmetries can improve this compilation process. We further investigate how, by analyzing the CHRs themselves we can find other opportunities for improving compilation, as well as determined

functional dependencies, symmetries and other algebraic features of the CHR constraints. The prototype HAL CHR compiler which applies these techniques produces highly efficient CHR executables.

Almost all of the optimizations considered in this paper are not specific to HAL, the optimizations that are not immediately applicable in a CHR compiler for Prolog are as follows. Mode information is not available for guards which means early guard scheduling may not be as effective, still assuming all variables are *invars* is safe and will account for most of the improvement. The determinism optimization +*det* in the experiments is not applicable since determinism declarations are not supported by Prolog systems.

There is substantial scope for further optimization of CHRs. These include: more complicated lookups (for example range lookups on tree indexes), replacing propagation rules by equivalent simplification rules, common subexpression elimination, unfolding of CHRs, and determining invariant information for stored constraints. We plan to continue improving the HAL CHR compiler to take advantage of these possibilities.

References

Abdennadher, S. (1997). Operational semantics and confluence of constraint propagation rules. *Pages 252–266 of:* Smolka, Gert (ed), *Proceedings of the Third International Conference on Principles and Practice of Constraint Programming.*

Demoen, B., García de la Banda, M., Harvey, W., Marriott, K., & Stuckey, P.J. (1999). An overview of HAL. *Pages 174–188 of: Proceedings of the Fourth International Conference on Principles and Practices of Constraint Programming.*

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

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

Holzbaur, C., Stuckey, P.J., García de la Banda, M., & Jeffery, D. (2001). Optimizing compilation of constraint handling rules. *Pages 74–89 of:* Codognet, P. (ed), *Logic Programming: Proceedings of the 17th International Conference.* LNCS. Springer-Verlag.

JCK. (2002). *JCK: Java constraint kit.* http://www.pms.informatik.uni-muenchen.de/software/jack/index.html.

Somogyi, Z., Henderson, F., & Conway, T. (1996). The execution algorithm of Mercury: an efficient purely declarative logic programming language. *Journal of Logic Programming*, **29**, 17–64.

A Building a constraint solver in HAL using CHRs

The program in Figure A 1 defines a simple bounds propagation solver for integers using constraint handling rules. From a HAL perspective it is a solver module defining a solver on the type `cint`. Line (*L3*) is the type definition for the new type `cint` which is a wrapped integer. The integer is used as a variable number. The integer is wrapped so that we have a new type that we can (re-)define equality for.The type is exported abstractly hence its definition is not visible outside the module, thus restricting operations on `cint` to those in this module.

Line (*L4*) is a re-instantiation declaration, which is required because we are going to

```
:- module interval.                                         (L1)
:- import int.                                              (L2)
:- export_abstract typedef cint -> f(int).                  (L3)
:- reinst_old cold = ground.                                (L4)
:- modedef cno -> (new -> cold).                            (L5)
:- modedef coo -> (cold -> cold).                           (L6)
:- VarNum glob_var int = 0.                                 (L7)

:- export pred init(cint).                                  (L8)
:-       mode init(cno) is det.                             (L9)
init(V) :- V = f($VarNum), $VarNum := $VarNum + 1,
         (bounds(V,-10000,10000) -> true ; error("not det.")).


:- export_only pred cint=cint.                              (L10)
:-       mode coo=coo is semidet.                           (L11)
X = Y :- eq(X,Y).

:- export chrc bounds(cint,int,int).                        (L12)
:-       mode bounds(coo,in,in) is semidet.                 (L13)
non_empty@  bounds(X,L,U) ==> U >= L.
redundant@  bounds(X,L1,U1) \ bounds(X,L2,U2) <=>
               L1 >= L2, U2 >= U1 | true.
intersect@bounds(X,L1,U1), bounds(X,L2,U2) <=>
               bounds(X,max(L1,L2),min(U1,U2)).

:- chrc eq(cint,cint).                                      (L14)
:- mode eq(in,in) is semidet.                               (L15)
equals @  eq(X,Y), bounds(X,LX,UX), bounds(Y,LY,UY) ==>
               bounds(Y,LX,UX),bounds(X,LY,UY).

:- export chrc cint >= cint.                                (L16)
:-       mode coo >= coo is semidet.                        (L17)
geq @     X >= Y, bounds(X,LX,UX), bounds(Y,LY,UY) ==>
               bounds(Y,LX,UY), bounds(X,LX,UY).

:- export chrc cint != cint.
:-       mode coo != coo is semidet.
neqset @  X != Y \ X != Y <=> true.
neqsym @  X != Y ==> Y != X.
neqlower@ X != Y,bounds(X,VX,VX),bounds(Y,VX,UY)==>bounds(Y,VX+1,UY).
nequpper@ X != Y,bounds(X,VX,VX),bounds(Y,LY,VX)==>bounds(Y,LY,VX-1).

:- export func cint + cint --> cint.
:-       mode coo + coo --> oo is semidet.
X + Y --> Z :- plus(X,Y,Z).

:- chrc plus(cint,cint,cint).
:- mode plus(in,in,in) is semidet.
plus@ plus(X,Y,Z),bounds(X,LX,UX),bounds(Y,LY,UY),bounds(Z,LZ,UZ)==>
     bounds(X,LZ-UY,UZ-LY),bounds(Y,LZ-UX,UZ-LX),bounds(Z,LX+LY,UX+UY).
```

Fig. A 1. A simple integer bounds propagation solver using CHRs

treat `cint`s in two ways. The `reinst_old` declares a new instantiation `cold` (to be associated with the `cint` type) which is equivalent to "`old`" (i.e. a possibly non-ground term) outside the module, and equivalent to `ground` inside the module. We require this because outside the module we treat `cint`s as bounds propagation solver variables, whereas inside the module they will be manipulated as wrapped integers (which are `ground`). Lines $(L5) - (L6)$ give two common modes of usage for `cint`s.

Variable numbers (for new `cint`s) are kept track of in a global integer counter `VarNum`. This is declared in line $(L7)$ with its type `int`, and initial value (0).

For any solver type we need to define two predicates `init/1` which initializes a new variable and `=/2` for equating two solver variables. Line $(L8)$ is the predicate declaration for `init/1` which is exported. Its mode is given in line $(L9)$, the mode `cno` takes a new object and returns a `cold` object (`old` outside this module, and `ground` inside this module). Its definition in the next line simply returns the wrapped counter value, and increments the counter. The predicate must always succeed exactly once hence its determinism is `det`, but to pass determinism checking the call to `bounds/3` is wrapped in an if-then-else (since the compiler cannot determine that it will not fail).

The `=/2` predicate is defined in line $(L10)$ as `export_only`, which makes it visible outside the module, but not visible inside the module. This is to avoid confusion with the equality on the internal view of `cint`s which simply treats them as ground terms rather than integer variables. Its mode definition in line $(L11)$ takes two `cold cint`s as input and returns the same instantiation (the `coo` mode). It may fail, so the determinism is `semidet`. Note the definition of `=/2` is made in terms of the non-exported constraint `eq/2`.

Finally we arrive at our first constraint. Line $(L12)$ defines an exported constraint `bounds/3` which relates a `cint` to two `int`s. The mode declaration on line $(L13)$ declares that the constraint must be invoked with an `old cint` and two ground integers. The CHR `non_empty` is a simple propagation rule. Note the advantages of a typed language, the `>=` on the right hand side is integer comparison, not to be confused with the constraint `>=` defined on line $(L16)$. The CHR `redundant` removes redundant bounds constraints. The CHR `intersect` replaces two `bounds/3` constraints on the same variable by one. Note that `bounds/3` occurs in many of the rules in the program not just the two defined immediately below its declaration.

When compiling the module `interval` we can determine the functional dependencies: $bounds(X, L, U) :: X \rightsquigarrow L$ and $bounds(X, L, U) :: X \rightsquigarrow U$, the symmetries $eq(X, Y) \equiv eq(Y, X)$, $neq(X, Y) \equiv neq(Y, X)$ and $plus(X, Y, Z) \equiv plus(Y, X, Z)$ and that each of the CHR constraints has a set semantics.

The lookups required for the program are (after reduction by functional dependencies): $bounds(X, \_, \_)$, $eq(X,)$, $eq(\_, Y)$, $neq(X, Y)$, $neq(X,)$, $neq(\_, Y)$, $plus(X, \_, \_)$, $plus(\_, Y, \_)$ and $plus(\_, \_, Z)$. Symmetry eliminates the indexes $eq(\_, Y)$, $neq(\_, Y)$ and $plus(\_, Y, \_)$.