Optimizing Compilation of Constraint Handling Rules

Christian Holzbaur[1], María García de la Banda[2], David Jeffery[2], and Peter J. Stuckey[3]

[1] Dept. of Medical Cybernetics and Art. Intelligence, University of Vienna, Austria
christian@ai.univie.ac.at

[2] School of Comp. Sci. & Soft. Eng., Monash University, Australia
{mbanda,dgj}@csse.monash.edu.au

[3] Dept. of Comp. Sci. & Soft. Eng., University of Melbourne, Australia
pjs@cs.mu.oz.au

Abstract. 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 discuss the optimizing compilation of Constraint Handling Rules (CHRs). In particualr, 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. 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 information about functional dependencies, symmetries and other kinds of information supporting optimizations.

1 Introduction

Constraint handling rules [3] (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. [4]). Efficient implementations are already available for the languages SICStus Prolog and Eclipse Prolog, and will soon appear for others such as Java [5] and HAL [2]

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 papers [4] on CHR compilation are:

- 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 were 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 many CHR solvers only use ground constraints this is an important issue.

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

$$\begin{array}{rl} \textit{simplification} & [\textit{name}@]\ c_1, \ldots, c_n \iff g \mid d_1, \ldots, d_m \\ \textit{propagation} & [\textit{name}@]\ c_1, \ldots, c_n \implies g \mid d_1, \ldots, d_m \\ \textit{simpagation} & [\textit{name}@]\ c_1, \ldots, c_l \setminus c_{l+1}, \ldots, c_n \iff g \mid d_1, \ldots, d_m \end{array}$$

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 $g \equiv \mathit{true}$.

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.

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. [1]. Although CHRs have a logical reading (see e.g. [3]) and programmers are encouraged to write confluent CHR programs, there are applications where a predictable order of rule applications is important. Hence, their textual order is used to resolve rule applicability conflicts in favor of earlier rules.

In this paper we focus on the implementation of CHRs in a programming language, such as HAL [2], 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)
:- export constraint gcd(int).                        (L3)
:- mode gcd(in) is det.                               (L4)
base @ gcd(0) <=> true.                               (L5)
pair @ gcd(N) \ gcd(M) <=> M >= N | gcd(M-N).         (L6)
```

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 ($L3$) exports the CHR constraint `gcd/1` which has one argument, an `int`. This is the *type* declaration for `gcd/1`. Line ($L4$) 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 [2, 6].

Lines ($L5$) and ($L6$) are the two 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 and determinism information means that CHR constraints can only have declared modes that do not change the instantiation state of their arguments,[1] since the compiler will be unable to statically determine when rules fire. The same restriction applies to dynamically scheduled goals in HAL (see [2]).[2]

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.

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

[2] Unlike dynamically scheduled goals in HAL, CHR constraints can have `multi` or `nondet` determinism.

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

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

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 this code together to form the entire program for the constraint. A more detailed description of the overall process is given in [4], which is the basis for the SICStus Prolog version of CHRs. In the rest of 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 an optimal 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. 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 $Partners$. 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 cost of an individual join is defined as a pair: the number of arguments in the new partner which are unfixed before the join; followed by the (negative of) the number of arguments which are fixed before the join. *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.

```
measure(Fixed,Partners,FDs,Guards)
  Lookups := ∅; Goal := true; score := (0,0); sum := (0,0)
  while true
    repeat
      Fixed0 := Fixed
      foreach g ∈ Guards
        if invars(g) ⊆ Fixed
          Goal := Goal, g; Fixed := Fixed ∪ outvars(g); Guards := Guards \ {g}
    until Fixed = Fixed0
    if Partners = ∅ return (score, Goal, Lookups)
    let Partners ≡ p(x̄), Partners1
    Partners := Partners1
    FDp := {p(x̄) :: fd ∈ FDs}
    Fixedp := fdclose(Fixed, FDp)
    fixedx := x̄ ∩ Fixedp
    cost := (|x̄ \ fixedx|, −|fixedx|)
    score := score + sum + cost; sum := sum + cost
    Lookups := Lookups ∪ {p((x_i ∈ fixedx ? x_i : _) | x_i ∈ x̄)}
    Fixed := Fixed ∪ x̄
    Goal := Goal, p(x̄);
  endwhile
```

Fig. 1. Algorithm for evaluating join ordering

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.

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 procedure. We also assume that each guard has exactly one mode (it is straightforward to extend the approach to multiple modes and more complex instantiations). 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 $\mathsf{fdclose}(Fixed, FDs)$ closes a set of fixed variables $Fixed$ under the functional dependencies. $\mathsf{fdclose}(Fixed, FDs)$ 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 $x \in F$.

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 scores calculated for the left-to-right partner order illustrated in the rule are $(3, -1)$, $(0, 0)$, $(0, -2)$, $(0, -1)$ for a total cost of $(12, -9)$[3] together with goal

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

[3] Note that the cost of `r(X,X,U)` is $(0, -2)$ because `W = U + 1` is executed before `r(X,X,U)` thus grounding `U`. Also note that `X` is counted only once.

and lookups `q(Y,_,_,_)`, `flag`, `r(X,X,U)`, `s(W)`. An optimal order has cost $(5,-7)$ 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 optimal order has 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)`.

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. In practice, we have not required this as left hand sides of CHRs are usually 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.[4]

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 some extra emphasis on indexes for ground data:

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

- lookup generalization: rather than build specialized indexes for lookups that share variables we simply use more general indexes. Thus, we replace any lookup $\mathtt{p}(v_1,\ldots,v_k)$ where v_i and v_j are the same variable by a lookup $\mathtt{p}(v_1,\ldots,v_{j-1},v'_j,v_{j+1},\ldots,v_k)$ 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 `eq(X,X)` can use the lookup for `eq(X,Y)`, followed by the guard `X = Y`.
- functional dependency reduction: we can use functional dependencies to reduce the requirement for indexes. We can replace any lookup $\mathtt{p}(v_1,\ldots,v_k)$ where there is a functional dependency $p(x_1,\ldots,x_k) :: \{x_{i_1},\ldots,x_{i_m}\} \leadsto x_j$

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

and $v_{i_1}, \ldots, v_{i_m}$ are input (as opposed to anonymous) variables to the query by the lookup $\mathtt{p}(v_1, \ldots, v_{j-1}, \_, v_{j+1}, \ldots, v_k)$. 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 $bounds(X, L, U) :: X \leadsto L$, the lookup `bounds(X,L,_)` can be replaced by `bounds(X,_,_)`.
- symmetry reduction: if $\mathtt{p}/k$ is symmetric on arguments i and j we have two symmetric lookups $\mathtt{p}(v_1, \ldots, v_i, \ldots, v_j, \ldots, v_k)$ and $\mathtt{p}(v'_1, \ldots, v'_i, \ldots, v'_j, \ldots, v'_k)$ 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. The default choice is a balanced binary search tree (BST). Note that using a BST we can sometimes merge two indexes, for example, a BST for `eq(X,Y)` can also efficiently answer `eq(X,_)` queries.

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.[5] We can use functional dependencies to determine when this multiset can have at most one element. This is the case for a lookup $\mathtt{p}(v_1, \ldots, v_k)$ with fixed variables $v_{i_1}, \ldots, v_{i_m}$ such that $\mathsf{fdclose}(\{x_{i_1}, \ldots, x_{i_m}\}, FDp) \supseteq \{x_1, \ldots, x_k\}$ where FDp are the functional dependencies for $\mathtt{p}/k$. For example, the lookup `bounds(X,_,_)` returns at most one constraint given the functional dependencies: $bounds(X, L, U) :: X \leadsto L$ and $bounds(X, L, U) :: X \leadsto U$. Iterators with at most one entry can return a `yesno` iterator rather than a list.

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.

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

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

[5] Some complexities arise with the insertion and deletion of constraints *during* the execution of the iterator, because once a rule commits, the changes to the store regarding the removal of constraints and the addition of the active constraint have to take effect to implement an "immediate update view".

```
gcd_3(M,CN1) :-
    (gcd_index_exists_iteration(N,CN2),
    M >= N, CN1 != CN2 -> %% guard
        gcd_delete_constraint(M,CN1),
        gcd(M-N), %% RHS
        gcd_3_succ_cont(M,CN1)
    ;   gcd_3_fail_cont(M,CN1) ).

gcd_2(N,CN1) :-
    gcd_index_iteration_init(I0),
    gcd_2_forall_iterate(N,CN1,I0).
```

```
gcd_2_forall_iterate(N,CN1,I0) :-
    gcd_iteration_last(I0),
    gcd_2_succ_cont(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(M-N) %% RHS
    ;   true), %% rule did not apply
    gcd_2_forall_iterate(N,CN1,I1).
```

Fig. 2. Code for existential partner search and universal partner search.

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.

Example 3. Consider the compilation of the 3rd occurrence of a `gcd/1` constraint in the program in the introduction (the second occurrence in ($L6$)) 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 2. 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 ($L6$)) requires universal search for partners. The compilation produces

```
gcd(N) :-                                  gcd_1_succ_cont(_,_).
    new_constraint_number(CN1),            gcd_1_fail_cont(N,CN1) :- gcd_3(N,CN1).
    gcd_insert_constraint(N,CN1),
    gcd_1(N,CN1).                          gcd_3_succ_cont(_,_).
gcd_1(N,CN1) :-                            gcd_3_fail_cont(N,CN1) :- gcd_2(N,CN1).
    (N = 0 -> %% Guard
        gcd_delete_constraint(N,CN1),      gcd_2_succ_cont(_,_).
        true, %% RHS                       gcd_2_fail_cont(_,_).
        gcd_1_succ_cont(N,CN1)
    ;   gcd_1_fail_cont(N,CN1)).
```

Fig. 3. Initial code, code for first occurrence and continuation code for `gcd`.

the code `gcd_2` shown in Figure 2. The predicate `gcd_index_iteration_init`, 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.

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 code is ordered according to the textual order of the associated constraint occurrences 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 $\mathtt{p}/k$ is as follows: Code for $\mathtt{p}/k$ creates a new constraint number and calls the first occurrence of code $\mathtt{p\_}o_1/k+1$. The fail continuation for $\mathtt{p\_}o_j/k+1$ is set to $\mathtt{p\_}o_{j+1}/k+1$. The success continuation for $\mathtt{p\_}o_j/k+1$ is also set to $\mathtt{p\_}o_{j+1}/k+1$ 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 4. 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 success continuation of `gcd_2` is `true` since it is last. The remaining code for `gcd/1` is given in Figure 3.[6]

4 Improving CHR compilation

In the previous section we examined the basics steps for compiling CHRs taking advantage of type, mode, functional dependency and symmetries information. In this section we explore other kinds of optimizations based on analysis of CHRs.

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

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. We can use whatever reasoning we please to prove the implication. Currently, both the SICStus and HAL version of the CHR compiler use very simple implication reasoning about identical constraints and `true`.

Example 5. 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 `intersect` 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.

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 6. 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 information can be inferred with a simple pre-analysis. For simplicity, we can consider a rule as *rhs-affects-store* if its right-hand-side calls a CHR constraint, or a local predicate which calls constraints (directly or indirectly), or (to be safe) an external predicate which is not a library predicate.

4.3 Set semantics

Although CHRs use a multiset semantics, often the constraints defined by CHRs have a set semantics, where the number of copies of a constraint does not matter. In the HAL version, indexes for constraints with set semantics can take advantage of this information (by not worrying about duplicates). We can recognize constraints with set semantics in two different ways.

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

$$p(x_1, \ldots, x_k) \setminus p(y_1, \ldots, y_k) \iff g \mid true$$

such that $\models x_1 = y_1 \wedge \cdots x_k = y_k \rightarrow \bar{\exists}_{\bar{x} \cup \bar{y}} g$ which occurs before any rule requires `p`/k to be stored or which can match two identical copies of `p`/k. For instance, the rule `red` from Example 5 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).

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

- there are no rules which can match two identical copies of `p`/k
- there are no rules that delete a constraint `p`/k without deleting all identical copies.
- there are no rules with occurrences of `p`/k that can generate constraints (on the rhs) which do not have set semantics.

A simple fixpoint analysis can detect such constraints starting from the assumption that all constraints have set semantics.

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.

Example 7. 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.

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 they encode functions using relations. Suppose p/k need not be stored before occurrences in a rule of the form

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

where $x_i, 1 \leq i \leq l$ and $y_i, z_i, l+1 \leq i \leq k$ are distinct variables. Then, this rule ensures that there is at most one constraint in the store of the form $p(x_1,\ldots,x_l,\_,\ldots,\_)$ at any time. 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 5 illustrates the functional dependencies $bounds(X,L,U) :: X \rightsquigarrow L$ and $bounds(X,L,U) :: X \rightsquigarrow U$.

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.

Example 8. The second rule for gcd/1 written twice illustrates the functional dependency $\texttt{gcd}(N) :: \emptyset \rightsquigarrow N$ 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 9. 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 an example.

Example 10. The rule in Example 7 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.

6 Experimental Results

Our initial version of the HAL CHR compiler performs only some of the optimizations outlined above, including join ordering and continuation optimization and late storage. We do not yet have the automated analysis to support discovery of functional dependencies, set semantics and symmetries, nor specialized indexes (which rely on this information). It is ongoing work to improve the compiler, to perform the appropriate analyses, and make use of the information during the compilation.

To get an estimate on the benefits achievable through the optimizing compilation, we have modified the code produced by the current compiler by hand, in a way as close as possible to how we foresee its implementation. The comparisons against the SICStus CHR versions primarily serve as a simple "reality check" for the HAL version in its infancy. Any deductions beyond that would have to consider all the differences between SICStus and HAL producing Mercury code.

We compare the performance on 3 small programs:

- `gcd` as described in the paper, where the query (a,b) is `gcd(`a`),gcd(`b`)`.
- `interval`: a simple bounds propagation solver on 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).
- `dfa`: a visual parser for 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 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:

Benchmark	Query	Orig	+yesno	+det	+nn	hand	SICS
gcd	(5000000,3)	1912	1592	529	222	192	10271
gcd	(10000000,3)	7725	6766	1596	666	574	20522
gcd	(15000000,3)	11471	10216	2129	891	766	30770

Benchmark	Query	Orig	+tree	+sym	+eq	SICS	SICS_v
interval	(10,1)	1870	550	620	600	17491	1298
interval	(12,1)	9400	1710	1860	1890	72950	3725
interval	(14,1)	124000	19100	20765	20520	876550	32318
interval	(10,2)	2460	840	890	605	21834	1656
interval	(12,2)	12495	2710	2810	1850	92475	4795
interval	(14,2)	165595	32270	35375	20535	1115910	42382
dfa	180	57	56	54	=	57435	=
dfa	300	126	98	88	=	180220	=
dfa	3000	10576	1209	1061	=	too long	=

Table 1. Execution times (ms) for various optimized versions of CHR programs.

```
circle(C1,R1), circle(C2,R2) \
line(P1,P2), line(P3,P2), line(P4,P2), 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).
```

The query a finds a (constant) small DFA (of 10 elements) in a large set of a graphical elements (to illustrate indexing).

The results are shown in Table 1. All timings are the average over 20 runs on a dual Pentium II-400MHz with 384M of RAM running under Linux RedHat 5.2 with kernel version 2.2, and are given in milliseconds. SICStus Prolog 3.8.4 is run under compact code (there is no fastcode for Linux).

For `gcd` we first give times for the original output of the compiler *Orig*. In the version +*yesno* the list storage of constraints is replaced by a +*yesno* structure (using the functional dependency). We can see a significant improvement here by just avoiding some loop overhead. Next in +*det* the determinism of produced code is modified to take into account the functional dependency. Here we can really see the benefits of taking advantage of the functional dependency. Finally in +*nn* constraint numbers are completely removed (and this massively simplifies the resulting code). We also give *hand* which uses the code in Example 8 (as a lower bound on where optimization can reach), and *SICS* the original code in SICStus Prolog 3.8.4.

In the second experiment we give: the original code *Orig*, +*tree* where all list indexes have been replaced by 234 trees, +*sym* where symmetric constraints have the symmetry handled by indexes, and +*eq* where set semantics optimizations are applied (note that an = means the code is identical. i.e. there was no scope for the optimization). Finally, we compare with the SICStus Prolog CHR compiler *SICS*, and for the `interval` example, a nonground version of the program

SICS_v which uses attributed variable indexing (the other benchmarks involve only ground constraints).

The advantage of join ordering is illustrated by the difference between HAL and SICStus on `dfa`, the principle difference here is simply the better join ordering and early guard scheduling of HAL.

Adding indexes is clearly important when there are a significant number of constraints and effective lookups. In `dfa`, since there is only one indexed lookup, if the constraint stores are too small the overhead of the trees nullifies the advantages of the lookup. As the number of elements grows the advantages become clear.

Handling symmetry turned out to be disappointing. While it can reduce the number of indexes, it doubles their size and hence the possible benefit is limited. The overhead of managing symmetry in the index overwhelms the advantages when the constraint store is small, the advantages only becomes visible when the constraint store grows very large (`dfa` 3000). 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.

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 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.

References

1. S. Abdennadher. Operational semantics and confluence of constraint propagation rules. In *Proceedings of the Third International Conference on Principles and Practice of Constraint Programming*, pages 252–266, 1997.
2. 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.
3. T. Frühwirth. Theory and practice of constraint handling rules. *Journal of Logic Programming*, 37(1–3):95–138, 1998.
4. C. Holzbaur and T. Frühwirth. Constraint handling rules, special issue. *Journal of Applied Artificial Intelligence*, 14(4), 2000.
5. JACK: Java constraint kit. http://www.fast.de/~mandel/jack/.
6. Z. Somogyi, F. Henderson, and T. Conway. The execution algorithm of Mercury: an efficient purely declarative logic programming language. *Journal of Logic Programming*, 29:17–64, 1996.