Welcome to Constraint Handling Rules

Thom Frühwirth

University of Ulm,
Germany
`www.informatik.uni-ulm.de/pm/mitarbeiter/fruehwirth/`

Abstract. Constraint Handling Rules (CHR) is a declarative concurrent committed-choice constraint logic programming language consisting of guarded rules that transform multisets of relations called constraints until no more change occurs. As an introduction to CHR as a general-purpose programming language we present some small programs using different programming styles and discuss their properties.

1 Introduction

Constraint Handling Rules (CHR) [Frü98, FA03, AFE05, SF05, Frü08] has not only cut its niche as a special-purpose language for writing constraint solvers, but also has been employed more and more as a general-purpose language in computational logic, reasoning and beyond. This is because CHR can embed many rule-based formalisms and implement algorithms in a declarative yet effective way.

CHR was motivated by the inference rules that are traditionally used in computer science to define logical relationships and arbitrary fixpoint computations. Like automated theorem proving, CHR uses formulae to derive new information, but only in a restricted syntax (e.g., no negation) and in a directional way (e.g., no contrapositives) that makes the difference between the art of proof search and an efficient programming language.

CHR adapts concepts from term rewriting systems (TRS) [BN98] for program analysis, for properties such as confluence [AFM99] and termination (e.g. [Frü00]). Other influences for the design of CHR were the General Abstract Model for Multiset Manipulation (GAMMA) [BCM88] and, of course, production rule systems like OPS5 [BFKM85], but also integrity constraints and event-condition-action rules found in relational databases and in deductive databases.

Implementations of CHR are abundant now. CHR does not necessarily impose itself as a new programming language, but as a language extension that blends in with the syntax of its host language, be it Prolog, Lisp, Haskell, C or Java. In the host language, CHR constraints can be posted; in the CHR rules, host language statements can be included.

The example programs here illustrate different programming styles in CHR. This paper is based on some example programs of the author's upcoming book on CHR [Frü08].

2 Preliminaries

A *CHR program* P is a finite set of rules consisting of constraints. (We do not discuss declarations for CHR constraints here since they are implementation-specific.) There are three kinds of rules:

Simplification rule: $Name \ @ \ H \Leftrightarrow G \mid B$
`[Name '@'] H '<=>' [G '|'] B.`
Propagation rule: $Name \ @ \ H \Rightarrow G \mid B$
`[Name '@'] H '==>' [G '|'] B.`
Simpagation rule: $Name \ @ \ H_1 \backslash H_2 \Leftrightarrow G \mid B$
`[Name '@'] H1 '\' H2 '==>' [G '|'] B.`

Name is an optional, unique identifier of a rule, the *(multi-)head (lhs, left hand side)* H (or H_1 and H_2) is a non-empty conjunction of CHR constraints, the optional *guard* G is a conjunction of built-in constraints, and the *body (rhs, right hand side)* B is a goal. A *goal* is a conjunction of built-in and CHR constraints. If the guard is omitted from a rule, it means the same as "*true* |".

Built-in constraints are predefined by the host language, while CHR constraints are defined by CHR rules.

Declaratively, a CHR rule logically relates head and body provided the guard is true. A simplification rule means that the head is true if and only if the body is true. A propagation rule means that the body is true if the head is true. A simpagation rule `Head1 \ Head2 <=> Body` is logically equivalent to the simplification rule `Head1, Head2 <=> Head1, Body`.

Basically, rules are applied to an initial conjunction of constraints (syntactically, a goal) until exhaustion (saturation), i.e. until no more change happens. An initial goal is called *query*. The intermediate goals of a computation are stored in the so-called *(constraint) store*. A final goal (store), to which no more rule is applicable, is called *answer (constraint)* or *result (of the computation)*.

We describe here (sequential) implementations according to the *refined operational semantics* [DSdlBH04] of CHR. Parallel or experimental implementations may apply the rules in different ways, but of course still respect the *standard abstract operational semantics* [Abd97].

A CHR constraint is both *code* and *data*. Every time a CHR constraint is posted (added, called, executed, asserted, imposed) as part of a goal, it checks itself the applicability of the rules it appears in. Such a constraint is called (currently) *active*. One tries and applies rules in the order they are written in the program, i.e. top-down and from left to right.

An active constraint is code which is evaluated like a procedure call. If, at the moment, no rule is applicable that removes it, the active constraint becomes passive data in the constraint store. It is called *(currently) passive* (delayed, suspended, spleeping, waiting).

Passive constraints may be woken (reconsidered, resumed, re-executed) to become active code if the environment (context) changes, concretely if their arguments get more constrained. This is the case if a variable occurring in the constraint gets more constrained by a built-in constraint.

There are several computational phases when a CHR rule is *tried* (for application) and finally *applied (executed, triggered)* (then it *fires*). These phases correspond to the constituents of a rule, read from left to right:

Head Matching. For each rule, one of its head constraints is matched against the active constraint. Matching succeeds if the constraint is an instance of the head, i.e., the head serves as a pattern. If matching succeeded and the rule has more than one head constraint, the constraint store is searched for *partner* constraints that match the other head constraints. Head constraints are searched from left to right, except that in simpagation rules, the constraints to be removed are tried before the head constraints to be kept (this is done for efficiency reasons). If the matching succeeds, the guard is checked. If there are several head constraints that match the active constraint, the rule is tried for each such matching. Otherwise the next rule is tried.

Guard Checking. A guard is a precondition on the applicability of a rule. The guard is basically a test that either succeeds or fails. If the guard succeeds, the rule applies, one *commits* to it and it fires. Otherwise the next rule is tried.

Body Execution. If the firing rule is a simplification rule, the matched constraints are removed from the store and the body of the rule is executed by executing the constraints in the body from left to right. Similarly for a firing simpagation rule, except that the constraints that matched the head part preceding `'\'` are kept. If the firing rule is a propagation rule the body of the rule is executed without removing any constraints. It is remembered that the propagation rule fired, so it will not fire again with the same constraints. When the currently active constraint has not been removed, the next rule is tried. According to rule type, we say that CHR constraints matching some constraint in the head of the rule are either *kept* or *removed* constraints.

3 Multiset Transformation

The following simple algorithms are similar to the ones found in other rule-based approaches, namely production rule systems and the GAMMA model of computation, but in CHR the programs are more concise.

The General Abstract Model for Multiset Manipulation (GAMMA) framework employs a *chemical metaphor*. States in a computation are chemical solutions where floating molecules interact freely according to reaction rules. Reactions can be performed in parallel provided they involve disjoint sets of molecules. This is referred to as *logical parallelism or declarative concurrency.* We can model molecules as CHR constraints.

These programs consist essentially of one constraint for representing *active data.* Pairs of such constraints are rewritten by a single simplification rule. Often, the rule can be written more compactly as a simpagation rule where one of the constraints (the catalyst) is kept and the other is removed and possibly replaced by an updated one. Optimizing CHR compilers will compile this to an efficient in-place update instead of removing and adding constraints.

3.1 Minimum

We compute the minimum of a multiset of numbers n_i, given as a computation of the query `min(`n_1`), min(`n_2`),..., min(`n_k`)`. We interpret `min(`n_i`)` to mean that the number n_i is potentially the minimum, that it is a *candidate* for the minimum value.

```
min(N) \ min(M) <=> N=<M | true.
```

The simpagation rule takes two `min` candidates and removes the one with the larger value. It keeps going until only one, the smallest value, remains as single `min` constraint. The program illustrates the use of multi-headed rules instead of explicit loops or recursion for iteration over data. This keeps the code extremely compact and easy to analyse. The rule corresponds to the intuitive algorithm that when we are to find the minimum from a given a list of numbers, we just cross out larger numbers until one, the minimum, remains.

For example, this computation is possible (where constraints involved in a rule application are underlined)

$\underline{\texttt{min(1)}}$, $\underline{\texttt{min(0)}}$, `min(2)`, `min(1)`
$\underline{\texttt{min(0)}}$, $\underline{\texttt{min(2)}}$, `min(1)`
$\underline{\texttt{min(0)}}$, $\underline{\texttt{min(1)}}$
`min(0)`

Program Properties. We used the rule application order of the *refined semantics* of CHR implementations, where computation in a query proceeds from left to right. In the *abstract semantics*, any order of rule applications is allowed, for example also:

$\underline{\texttt{min(1)}}$, `min(0)`, $\underline{\texttt{min(2)}}$, `min(1)`
$\underline{\texttt{min(1)}}$, `min(0)`, $\underline{\texttt{min(1)}}$
$\underline{\texttt{min(0)}}$, $\underline{\texttt{min(1)}}$
`min(0)`

In the two examples above, the answer is the same. Actually, it is easy to see that the answer will always be the same, i.e. the minimum value, no matter in which order the rules are applied to which pair of constraints. We call this property *confluence.*

The rules can even be applied in *parallel* to different parts of the query.

$\underline{\texttt{min(1)}}$, $\underline{\texttt{min(0)}}$, $\underline{\texttt{min(2)}}$, $\underline{\texttt{min(1)}}$
$\underline{\texttt{min(0)}}$, $\underline{\texttt{min(1)}}$
`min(0)`

Obviously we arrive at the answer in less *computation steps.*

The program is obviously terminating, because the rule removes a CHR constraint and does not introduce new ones. Therefore the number of rule applications is one less than the number of `min` constraints. We can apply a rule in constant time. Given any two `min` constraints, we can always apply the rule - either in one pairing order or in the other. Therefore the complexity of this little program is linear in the number of `min` constraints, i.e. linear in the size of the initial goal.

We can also stop the computation at *any time* and observe the current store as intermediate answer. We can then continue by applying rules to this store without the need to recompute from scratch and no need to remember anything about how we arrived at the current store. If we stop again, we will observe another intermediate answer that is closer to the final answer than the one before. By closer we mean here that the store has less `min` constraints, i.e. less candidates for the final minimum. The intermediate answers more and more approximate the final answer. This property of a CHR program is called *anytime algorithm property.* Note that by this description, an anytime algorithm is also an *approximation algorithm.*

Now assume that while the program runs, we add a `min` constraint. It will eventually participate in the computation in that the rule will be applied to it. The answer will be correct, as if the newly added constraint had been there from the beginning but ignored for some time. This property of a CHR program is called *incrementality* or *online algorithm property.*

Guard Checking. So far we assumed that the `min` constraints contain given values. In that case, the guard acts as a test that compares two such values. But in general, under the *abstract standard semantics*, even though not necessarily in a given CHR implementation, the guard is made out of built-in constraints that hold if they are logically implied by the current *store.* While in current practical implementations of CHR, a guard check will give an instantion error or silently fail if unbound variables occur in it, the same guard check may succeed under the abstract semantics. For example, the query `min(A), min(B), A=<B` will reduce to `min(A), A=<B`, because we know that `A=<B` and that is exactly what the guard asks for. Similarily, the query `min(A), min(B), A<B` will reduce to `min(A), A<B`. Finally, the query `min(A), min(A)` will reduce to `min(A)`. But the query `min(A), min(B)` will not proceed, because we know nothing about the relationship of the unknown values `A` and `B`.

Now consider what happens if we modify the program in that we strenghten the guard. If we replace `N=<M` by `N<M`, *multiple occurrences* (*duplicates*) of the final minimum constraint will no longer be removed. If we replace `N=<M` by `N=M`, we will just remove duplicates. Both rules taken together have the same behavior as our initial rule, provided we work with known values.

```
min(N) \ min(M) <=> N<M | true.
min(N) \ min(M) <=> N=M | true.
```

If values are only partially known, it turns out the the two rules are weaker than the single initial rule. Consider the previous examples. Most of them still work, but the query `min(A), min(B), A=<B` will not reduce at all, because the built-in constraint `A=<B` is too weak to imply one of the guards of the two rules, `A<B` or `A=B`. We say that these two programs are not *operationally equivalent*, even though logically, they are. (The logical reading of rules as formulae is their declarative semantics [Abd97].)

Variations. If we want to use this rule for minimum in a larger program, we may be faced with some pragmatical issues. We may want to compute several minima from different sources and need to dinstinguish them. It suffices to add an identifier to the `min` constraint and modify the minimum rule so that it refers only to constraints with the same identifier:

```
min(Id,N) \ min(Id,M) <=> N=<M | true.
```

In general, this technique of adding an explicit identifier to each constraint can be used to localize computations, i.e. to implement *local constraint stores.*

3.2 Prime Numbers Sieve of Erastosthenes

We implement the algorithm known as Sieve of Erastosthenes, but without any particular sifting order. Given some numbers, the rule just removes multiples of each of the numbers.

```
sift @ prime(I) \ prime(J) <=> J mod I =:= 0 | true.
```

We give the rule a conjunction of prime number candidates consisting of all numbers from 2 up to `N`, i.e., `prime(2),prime(3),prime(4),...prime(N)`. The candidates react with each other such that each number absorbs multiples of itself. When we give it all integers up to a given bound starting from 2, all composite numbers will be removed after exhaustive application of the rule, so that only prime numbers remain.

For example, this computation is possible

$\texttt{prime(7), }\underline{\texttt{prime(6)}}\texttt{, prime(5), prime(4), }\underline{\texttt{prime(3)}}\texttt{, prime(2)}$
$\texttt{prime(7), prime(5), }\underline{\texttt{prime(4)}}\texttt{, prime(3), }\underline{\texttt{prime(2)}}$
$\texttt{prime(7), prime(5), prime(3), prime(2)}$

The `sift` rule is similar to the one for minimum in that it compares two numbers and removes one of them. But unlike minimum, the rule is not applicable to arbitrary pairs of prime number candidates.

As before, the program has the desirable properties that are typical for CHR. For example, the rule is obviously terminating, since it removes constraints without adding new ones.

Generating Numbers. To generate the prime number candidates, we may use an auxiliary CHR constraint `upto`[1]:

```
upto(1) <=> true.
upto(N) <=> N>1 | prime(N), upto(N-1).
```

To the same effect, we can use the `prime` constraint itself.

```
prime(N) ==> N>2 | prime(N-1).
```

Of course, this rule must come before the `sift` rule. Otherwise a prime number candidate may be removed before generating its predecessors.

[1] For readability, we use arithmetic expressions like `N-1` in arguments, while in Prolog, one may explicitely have to compute the result using `is/2`.

Both rule variants generate the prime candidates in descending order. Increasing order is preferable, because smaller prime candidates increase the chance that the `sift` rule is applicable. We can easily fix `upto` by exchanging the recursive call with the generation of the prime:

```
upto(N) <=> N>1 | upto(N-1), prime(N).
```

We cannot fix the variation using `prime` itself this way.

Primes Sieve in CHR for Java. The following code implements the three rules for primes in JCK, the first CHR implementation in Java. The syntax of CHR rules was chosen to be similar to that of the host language Java. For example, guards are not written between head and body of a rule, but as `if` expressions before the head. The rule name comes last. This illustrates that the concrete syntax of CHR is not fixed, but rather can be adapted to the host language.

```
handler primes { class java.lang.Integer; class IntUtil;

constraint prime(java.lang.Integer);
constraint upto(java.lang.Integer);

rules { variable java.lang.Integer N, M, I, J;

      {upto(1)} <=> {true} ;
      if (IntUtil.gt(N,1)) {upto(N)} <=>
                       {M=IntUtil.dec(N) && prime(N) && upto(M)};

      if (IntUtil.modNull(J,I)){prime(I) &\& prime(J)} <=>
                                                      {true} sift;
      }
}
```

A more recent implementation of CHR in Java, the K.U.Leuven JCHR system, uses the more traditional Prolog-style concrete syntax of CHR, which eases porting of code between Prolog and Java CHR systems.

```
handler primes {

    constraint upto(int);
    constraint prime(int);

rules { variable int N, I, J;

      upto(1) <=> true.
      upto(N) <=> IntUtil.gt(N,1)|prime(N), upto(intUtil.dec(N)).

      sift @ prime(I) \ prime(J) <=> intUtil.modZero(J,I) | true.
      }
}
```

4 Procedural Algorithms

We now employ a more tradional style of programming, where constraints are relations that resemble procedures as they are used in imperative programming languages. Results of a computation are not returned as constraints, but as values of variables that are bound. As we will already see with our first example of Fibonacci numbers, CHR supports different programming styles and it is easy to change between them.

4.1 Fibonacci

The n-th Fibonacci number is defined inductively as follows:

$$fib(0) = fib(1) = 1; fib(n) = fib(n-1) + fib(n-2) \text{ if } n \geq 2$$

When we implement this definition in CHR, we translate the functional notation of fib into relational notation, and the equivalence becomes a simplification rule.

Top-Down Evaluation. The CHR constraint `fib(N,M)` holds if the `N`-th Fibonacci number is `M`.

```
f0 @ fib(0,M) <=> M=1.
f1 @ fib(1,M) <=> M=1.
fn @ fib(N,M) <=> N>=2 | fib(N-1,M1), fib(N-2,M2), M is M1+M2.
```

The three rules are a direct translation of the definition. For example, the query `fib(8,A)` yields `A=89`, the query `fib(12,233)` succeeds, the query `fib(11,233)` fails, the query `fib(N,233)` delays.

As is well known, such a direct implementation has exponential time complexity because of the double recursion that recomputes the same Fibonacci numbers over and over again in different parts of the recursions.

Tabling and Memorization. We would like to store the results of Fibonacci numbers that we already have computed and look them up to avoid computing the same Fibonacci number several times. Since CHR constraints are both *operations and data*, it is easy to change the rules accordingly. We just have to turn the three simplification rules into propagation rules, so that the left hand side constraints are kept. In this way the result of the computation will be kept in the constraint store as data.

The rule for the look-up of already computed Fibonacci numbers has to come first, so that it is applied before we compute in the usual way using the expensive recursive definition.

```
mem @ fib(N,M1) \ fib(N,M2) <=> M1=M2.

f0 @ fib(0,M) ==> M=1.
f1 @ fib(1,M) ==> M=1.
fn @ fib(N,M) ==> N>=2 | fib(N-1,M1), fib(N-2,M2), M is M1+M2.
```

The rule `mem` for look-up enforces the functional dependency between input and output of the Fibonacci relation, in other words it uses the fact the `fib` defines a function. The query `fib(8,A)` now returns *all* Fibonacci numbers up to `8`: `fib1(0,1), fib1(1,1), fib1(2,2), ..., fib1(7,21), fib1(8,34)`.

The effect of memorization is dramatic: while the original rules have exponential complexity, the new version has only linear complexity, because each Fibonacci number is only computed once. When executed from left to right, the second recursive call is just a lookup using the `mem` rule. Actually, the `mem` rule does more than just looking up computed results, it in effect merges two computations that must have the same result into one, even if both computations are still ongoing. To see this, consider a query `fib(N,A)` with `N>=2`, where the `N`-th Fibonacci number is computed for the first time. The constraint `fib(N,A)` will thus try the `mem` rule in vain and finally the recursive rule `fn` will apply. Since it is a propagation rule, the constraint `fib(N,A)` will not be removed.

If the `N`-th Fibonacci number is called again, say with `fib(N,B)`, then the constraint `fib(N,B)` will try the `mem` rule, and there it will first try to match the constraint to the right of \ under the refined semantics. This succeeds and the old `fib(N,A)` is found as a *partner constraint.* The rule applies, the new `fib(N,B)` will be removed and instead the variables for the result will be equated using `A=B`.

Bottom-Up Evaluation. Another way of computing the Fibonacci numbers efficiently is by using only data and compute larger numbers from smaller ones. Basically, the idea is to reverse head and body of the rules.

```
fn @ fib(N1,M1), fib(N2,M2) ==> N2=:=N1+1 | fib(N2+1,M1+M2).
```

Since reversing the rules `f0` and `f1` gives ill-formed CHR rules (they do not have a head), we added the first two Fibonacci numbers in the query, `fib(0,1), fib(1,1)`. Of course, the resulting computation is infinite, and in order to observe the results, we have to add a rule in front such as:

```
fib(N,M) ==> write(fib(N,M)).
```

Note that if we are only interested in the Fibonacci numbers, we could drop the first arguments of `fib`.

The computation can be made finite by introducing an upper bound `Max`. The query `fib_upto(Max)` will produce all Fibonacci numbers up to `Max`. The constraint `fib_upto(Max)` is also used to introduce the first two Fibonacci numbers.

```
f01@ fib_upto(Max) ==> fib(0,1), fib(1,1).
fn @ fib_upto(Max), fib(N1,M1), fib(N2,M2) ==>
                          Max>N2, N2=:=N1+1 | fib(N2+1,M1+M2).
```

A version that is faster than any discussed so far can be achieved with a tiny change in the previous program: we turn the propagation rule into a simpagation rule that only keeps the (last) two Fibonacci numbers (we do not need more information to compute the next one).

```
fn @ fib_upto(Max), fib(N2,M2) \ fib(N1,M1) <=>
                        Max>N2, N2=:=N1+1 | fib(N2+1,M1+M2).
```

We have exchanged the order of the two `fib` constraints in the head so that the simpagation rule removes the smaller Fibonacci number.

Procedural Style Version. Since we now keep only the two last Fibonacci numbers, we can merge the three constraints of the head of the `fn` rule into one constraint, and the same for the three constraints that will be present after the rule has been applied (the two kept constraints from the head and the new one from the body). The resulting code is the most efficient:

```
f01@ fib_upto(Max) <=> fib(Max,1,1,1).
fn @ fib(Max,N,M1,M2) <=> Max>N | fib(Max,N+1,M2,M1+M2).
```

4.2 Newton's Method for Square Roots

Newton iteration is an approximation method for the value of polynomial expressions relying on derivates. We would like to compute the square root. As can be computed by Newton's method, the approximations for square roots are related by the formula $G_{i+1} = (G_i{+}X/G_i)/2$.

Since CHR programs already implement anytime, i.e. approximation algorithms, the implementation in CHR is straightforward. We assume that the answer is returned as a CHR constraint. `sqrt(X,G)` means that the square root of `X` is approximated by `G`. This rule computes the next approximation.

```
sqrt(X,G) <=> abs(G*G/X-1)>0 | sqrt(X,(G+X/G)/2).
```

The query is just `sqrt(GivenNumber,Guess)`. Both numbers must be positive, and if no guess is known, we may take `1`. The guard stops its application if the approximation is exact. Since this is unlikely in practice when floating point numbers are used and also to improve efficiency by avoiding iterations, we replace `0` in the guard by a sufficiently small positive number ϵ.

Since the quality of approximation is often in the eye of the beholder, we may implement a more interesting, *demand-driven* version of the algorithm. An approximation step is performed *lazily*, only on demand, which is expressed by the constraint `improve(Expression)`.

```
improve(sqrt(X)), sqrt(X,G) <=> sqrt(X,(G+X/G)/2).
```

Of course the constraint `improve` can be extended with a counter or combined with a check for the quality of the approximation.

5 Graph-Based Algorithms

5.1 Transitive Closure

Transitive closure is an essential operation that occurs in many algorithms, e.g. for graphs, in automated reasoning and inside constraint solvers. The transitive closure R^+ of a binary relation R is the smallest transitive relation that contains

R. The relation xR^+y holds iff there exists a finite sequence of elements x_i such that $xRx_1, x_1Rx_2, \ldots, x_{n-1}Rx_n, x_nRy$ holds.

For example, if R is the parent relation, then its transitive closure R^+ is the ancestor relation. If R is the relation of cities connected by direct trains, then its transitive closure also contains cities reachable by changing trains.

We can depict the relation R as a *directed graph*, where there is a *directed edge (arc)* from node (vertex) x to node y iff xRy holds. The transitive closure then corresponds to all paths in the graph. The *length of the path* is the number of edges in the path.

We implement the relation xRy as edge constraint `e(X,Y)` and its transitive closure xR^+y as path constraint `p(X,Y)`.

```
e(X,Y) ==> p(X,Y).
e(X,Y), p(Y,Z) ==> p(X,Z).
```

The implementation in CHR uses two propagation rules that compute the transitive closure *bottom-up*. In the first rule, for each edge, a corresponding path is added. The rule reads: If there is an edge from `X` to `Y` then there is also a path from `X` to `Y`. The second rule extends an existing path with an edge in front. It reads: If there is an edge from `X` to `Y` and a path from `Y` to `Z` then there is also a path from `X` to `Z`.

For example, the query `e(1,2), e(2,3), e(2,4)` adds the path constraints `p(1,4),p(2,4),p(1,3),p(2,3),p(1,2)`. Query `e(1,2), e(2,3), e(1,3)` will compute `p(1,3)` *twice*, because there are two ways to go from node `1` to node `3`, directly or via node `2`.

Termination. The program does not terminate with a cyclic graph. Consider the query `e(1,1)`, where infinitely many paths `p(1,1)` are generated by the second propagation rule. There are various compiler optimizations and options that avoid the repeated generation of the same constraint in this context, but here we are interested in a source-level solution that works in any implementation that follows the *refined semantics*.

Duplicate Removal. Termination can be restored easily by removing *duplicate* path constraints before they can be used. In other words, we would like to enforce a *set-based semantics* for path constraints. This is ensures termination, since in a given finite graph, there can only be a finite number of different paths. This *simpagation rule* removes duplicates:

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

The rule must come first in the program.

Single-Source Paths. We may specialize the transitive closure rules so that only paths that reach a given single target node are computed. We simply add the target node as a constraint:

```
target(Y), e(X,Y) ==> p(X,Y).
target(Z), e(X,Y), p(Y,Z) ==> p(X,Z).
```

However, this does not work if we want to fix the source node in the same way:

```
source(X), e(X,Y) ==> p(X,Y).
source(X), e(X,Y), p(Y,Z) ==> p(X,Z).
```

The reason is that in the second rule we need a path from `Y` to `Z` to be extended, but we only produce paths starting in `X`. If we exchange the edge and path constraints in the second rule so that we add an edge at the end of an existing path, then we can add a restriction to a source node as simply as before:

```
source(X), e(X,Y) ==> p(X,Y).
source(X), p(X,Y), e(Y,Z) ==> p(X,Z).
```

Shortest Path Lengths. Let us add an argument to the path constraint that holds the length of the path. When we adapt the duplicate removal rule, we keep the shorter path. This also ensures termination. The path propagated from an edge has length 1. A path of length n extended by an edge has length $n + 1$.

```
p(X,Y,N) \ p(X,Y,M) <=> N=<M | true.
e(X,Y) ==> p(X,Y,1).
e(X,Y), p(Y,Z,N) ==> p(X,Z,N+1).
```

For example, the query `e(X,X)` reduces to `p(X,X,1)`. For the query `e(X,Y), e(Y,Z), e(X,Z)`, the answer is
`e(X,Y), e(Y,Z), e(X,Z), p(X,Z,1), p(Y,Z,1), p(X,Y,1)`.

These rules can be easily generalized to compute shortest distances: replace `1` by the additional distance `D` given in the edge constraint `e`:

```
p(X,Y,N) \ p(X,Y,M) <=> N=<M | true.
e(X,Y,D) ==> p(X,Y,D).
e(X,Y,D), p(Y,Z,N) ==> p(X,Z,N+D).
```

5.2 Ordered Merging and Sorting

We use a binary CHR constraint written in infix notation, `A --> B`, to represent a *directed edge (arc)* from node `A` to node `B`. We use a *chain* of such arcs to represent a sequence of values that are stored in the nodes, e.g. `0-->2, 2-->5`.

Ordered Merging. We assume ordered chains with nodes in ascending order. So `A-->B` means that `A=<B`. We also say that `B` is the *immediate successor* of `A`.

The following one-rule program performs an ordered merge of two chains by zipping them together, provided they start with the same (smallest) node.

```
A --> B \ A --> C <=> A<B,B<C | B --> C.
```

Consider two arcs to which the rule applies. For example, consider the query `0-->2, 0-->5`. It will result in `0-->2, 2-->5` after one rule application. Basically we add the arc `B-->C` to represent `B<C`. Thus the arc `A-->C` now becomes redundant due to transitivity and is removed. This rule in a sense undoes *transitive closure*. It flattens out a *branch* in a graph.

The code basically works like a zipper. In the rule, `A` denotes the current position where there is a branch. During computation, all nodes up to `A` have already been merged, now the successors of `A` in the two chains are examined. The arc from `A` to `B`, the smaller of the two successor nodes of `A`, is kept, since `B` must be the immediate successor of `A`. The second arc is replaced by an arc from `B` to `C`. If the first chain is not finished yet, the new branch will be at `B` now. The rule applies again and again until there is no more branch left by using up at least one chain. (The chains can have different length.)

For example, the query `0-->2, 2-->5, 0-->3, 3-->7` will produce the answer `0-->2, 2-->3, 3-->5, 5-->7`. (Note that the constraints in the answer may not necessarily be sorted in that way.)

Termination and Correctness. Applying the rule will not change the number of arcs and the set of involved nodes, i.e. values. The nodes on the right of an arc will not change, too. Only a node on the left may be replaced by a larger node. Since the only rule replaces smaller node values by strictly larger ones without changing anything else and there is only a finite number of values, the program terminates. The application of the rule keeps the invariant that the two graphs are ordered chains.

We can prove correctness by contradiction: If there is an arc whose right node value is not the immediate successor of the left node value, then the chain is not ordered or disconnected. During computation the chains will share a longer and longer common prefix. If no rule is applicable, the two chains have been merged, there is only one chain, so that chain must be ordered, too.

Duplicate Removal. Note that duplicate values are ignored by the rule due to its guard, as they occur as arcs of the form `A-->A`. Also duplicate arcs of the form `A-->B, A-->B` are ignored. To remove duplicate values and duplicate arcs, we may add the two rules:

```
A --> A  <=> true.
A --> B \ A --> B <=> true.
```

The rule for duplicate arcs can be made redundant when we slightly generalize its guard of our initial merge rule:

```
A --> B \ A --> C <=> A<B, B=<C | B --> C.
```

Concretely, from `A-->B, A-->B`, where `A<B`, the sorting rules produces `A-->B, B-->B`. The arc `B-->B` will be removed by the rule for duplicate arcs.

Sorting. We can now perform an ordered merge of two chains that are in ascending order. But the merge rule also works with more than two chains. It will actually merge them simultaneously. Based on this observation, we can implement a merge sort algorithm. If we want to sort n values, we take n one length chains starting with the same smallest (dummy) value (in the example it is 0). Applied repeatedly to a left node, the merge rule will find its immediate successor. As before, the answer is a single, ordered chain of arcs.

In its generality, the code turns a certain type of ordered tree into an ordered chain. Actually, any graph of ordered arcs where all nodes can be reached from a single root node can be sorted. There are no duplicate nodes on the right of an arc, i.e., no right branches. The branches are on the left nodes of an arc, and they are removed by our sorting rule.

Our one-rule sorting program has quadratic complexity when the complier optimisation of *indexing* is used, an optimal lin-log complexity version is also possible with just one additional rule.

6 Conclusions

We have introduced CHR by presenting some small programs written in different programming styles. We also discussed the properties of these programs.

References

[Abd97] Abdennadher, S.: Operational Semantics and Confluence of Constraint Propagation Rules. In: Smolka, G. (ed.) CP 1997. LNCS, vol. 1330. Springer, Heidelberg (1997)

[AFE05] Abdennadher, S., Frühwirth, T., Holzbaur, C. (eds.): Special Issue on Constraint Handling Rules. Journal of Theory and Practice of Logic Programming (TPLP) (to appear, 2005)

[AFM99] Abdennadher, S., Frühwirth, T., Meuss, H.: Confluence and Semantics of Constraint Simplification Rules. Constraints 4(2), 133–165 (1999)

[BCM88] Banâtre, J.-P., Coutant, A., Le Metayer, D.: A Parallel Machine for Multiset Transformation and its Programming Style. Future Generation Computer Systems 4(2), 133–144 (1988)

[BFKM85] Brownston, L., Farrell, R., Kant, E., Martin, N.: Programming Expert Systems in OPS5: An Introduction to Rule-based Programming. Addison-Wesley, Boston (1985)

[BN98] Baader, F., Nipkow, T.: Term Rewriting and All That. Cambridge Univ. Press, Cambridge (1998)

[DSdlBH04] Duck, G.J., Stuckey, P.J., de la Banda, M.G., Holzbaur, C.: The Refined Operational Semantics of Constraint Handling Rules. In: Demoen, B., Lifschitz, V. (eds.) Proceedings of the 20th International Conference on Logic Programming (2004)

[FA03] Frühwirth, T., Abdennadher, S.: Essentials of Constraint Programming. Springer, Heidelberg (2003)

[Frü98] Frühwirth, T.: Theory and Practice of Constraint Handling Rules, Special Issue on Constraint Logic programming. Journal of Logic Programming 37(1-3), 95–138 (1998)

[Frü00] Frühwirth, T.: Proving Termination of Constraint Solver Programs. In: Apt, K.R., Kakas, A.C., Monfroy, E., Rossi, F. (eds.) Compulog Net WS 1999. LNCS, vol. 1865, pp. 298–317. Springer, Heidelberg (2000)

[Frü08] Frühwirth, T.: Constraint Handling Rules. Cambridge University Press, Cambridge (to appear, 2008)

[SF05] Schrijvers, T., Frühwirth, T.: CHR Website (May 2005), `http://www.cs.kuleuven.ac.be/~dtai/projects/CHR/`