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 is applied. If the guard fails, the active constraint tries the next head matching.

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

1.3 Exercises

Compare the following CHR programs, which consist each of *one* of the given rules, based on their answers to the suggested queries.

Exercises

1.1 Single-headed rules:

```
p <=> q.
p ==> q.
```

Multi-headed rules:

```
p , q <=> true.
p \ q <=> true.
```

Use some queries containing `p` and/or `q`.

1.2 Simplification rules with logical variables and syntactic equality:

```
p(a) <=> true.
p(X) <=> true.
p(X) <=> X=a.
p(X) <=> X=a | true.
p(X) <=> X==a | true.
```

Propagation rules with logical variables and syntactic equality:

```
p(a) ==> true.
p(X) ==> true.
p(X) ==> X=a.
p(X) ==> X=a | true.
p(X) ==> X==a | true.

p(X) ==> q(X).
p(a) ==> q(X).
p(a) ==> q(a).
p(X) ==> q(Y).
```

Queries: (a) `p(a)`, (b) `p(b)`, and (c) `p(C)`.

1.3 Arithmetic comparison:

```
p(X) <=> X>1 | q(X).
p(X) ==> X>1 | q(X).
p(X) ==> X>1 | fail.
p(X) ==> X=<1.
```

Queries: (a) `p(0)`, (b) `p(1)`, (c) `p(2)`, and (d) `p(A)`.

1.4 Multi-headed rules, conjunctions in head and body, unary constraints:

```
p(X), q(X) <=> r(X).
p(X), q(Y) <=> X==Y | r(X).
p(X), q(Y) <=> X=Y, r(X).
p(X) \ q(X) <=> r(X).
```

Queries: (a) `p(a), q(a)`, (b) `p(a), q(b)`, (c) `p(a), q(a), q(b)`, and (d) `p(a), q(b), q(a)`.

1.5 Multi-headed rules, conjunctions in head and body, binary constraints:

```
c1 @ c(X), c(X) <=> q(X,X).
c2 @ c(X), c(Y) <=> r(X,Y).
c3 @ c(X), c(X) ==> q(X,X).
c4 @ c(X), c(Y) ==> r(X,Y).
```

Queries: a) `c(a)`, b) `c(a), c(a)`, c) `c(a), c(b)`, d) `c(X), c(X)`, e) `c(X), c(Y)`, and f) `c(X), c(Y), X=Y`

Selected Answers:

```
c1 @ c(X), c(X) <=> q(X,X).
a) q(X,X)
b) c(X), c(Y)
```

```
c) Y = X, q(X,X)

c2 @ c(X), c(Y) <=> r(X,Y).
a) r(X,X)
b) r(Y,X)
c) Y = X, r(X,X)

c3 @ c(X), c(X) ==> q(X,X).
a) c(X), c(X), q(X,X), q(X,X)
b) c(X), c(Y)
c) Y = X, c(X), c(X), q(X,X), q(X,X)

c4 @ c(X), c(Y) ==> r(X,Y).
a) c(X), c(X), r(X,X), r(X,X)
b) c(X), c(Y), r(Y,X), r(X,Y)
c) Y = X, c(X), c(X), r(X,X), r(X,X)
```

1.6 Binary constraints:

```
q1 @ p(X,Z), q(Z,Y) <=> q(X,Y).
q2 @ q(Z,Y), p(X,Z) <=> q(X,Y).
q3 @ p(X,Z), q(Z,Y) ==> q(X,Y).
q4 @ q(Z,Y), p(X,Z) ==> q(X,Y).
q5 @ p(X,Z) \ q(Z,Y) <=> q(X,Y).
q6 @ q(Z,Y) \ p(X,Z) <=> q(X,Y).
```

Queries: a) `p(a,b), q(b,c)`, b) `p(A,B), q(B,C)`, c) `p(A,B), q(B,C), p(D,A)`, d) `p(X,C), p(Y,C), q(C,A)`, and e) `p(Y,C), p(X,C), q(C,A)`.

Selected Answers:

```
q1 @ p(X,Z), q(Z,Y) <=> q(X,Y).
b) q(A,C)
c) q(D,C)

q2 @ q(Z,Y), p(X,Z) <=> q(X,Y).
b) q(A,C)
c) q(D,C)

q3 @ p(X,Z), q(Z,Y) ==> q(X,Y).
b) p(A,B), q(B,C), q(A,C)
c) p(A,B), q(B,C), q(A,C), p(D,A), q(D,C)
```

```
q4 @ q(Z,Y), p(X,Z) ==> q(X,Y).
b) p(A,B), q(B,C), q(A,C)
c) p(A,B), q(B,C), q(A,C), p(D,A), q(D,C)

q5 @ p(X,Z) \ q(Z,Y) <=> q(X,Y).
b) p(A,B), q(A,C)
c) p(A,B), p(D,A), q(D,C)

q6 @ q(Z,Y) \ p(X,Z) <=> q(X,Y).
b) q(B,C), q(A,C)
c) q(B,C), q(A,C), q(D,C)
```

1.4 Origins and Applications of CHR

CHR [Frü98, HF00, FA03, AFH05, FMS06, SWSK08] has many roots and combines their features in an attractive way, enabling powerful applications.

Origins. Prolog and Logic programming (LP) [Kow86, CR93], constraint logic programming (CLP) [Hen91, JM94, JL87, MS98, FA03, RBW06] and concurrent committed-choice logic programming (CC) [Mah87, Ued88, Sha89, Sar93] are direct ancestors of CHR. CHIP was the first CLP language to introduce feasible constructs (demons, forward rules, conditionals) [DHS^{+}88, Hen89] for user-defined constraints. These various constructs have been generalized into and made uniform by CHR.

CHR adapts concepts from term rewriting systems (TRS) [BN98] for program analysis. Augmented term rewriting was used in the functional language Bertrand [Lel88] to implement constraint-based algorithms.

Other influences for the design of CHR were the General Abstract Model for Multiset Manipulation (GAMMA) [BCM88, BM93], the Chemical Abstract Machine (CHAM) based on it [BB92], and, of course, production rule systems like OPS5 [BFKM85], but also integrity constraints and event-condition-action rules found in relational and deductive database systems.

Executable rules with multiple head constraints were also proposed in the literature to model parallelism and distributed agent processing as well as logical objects [BCM88, AP90] and for constraint solving [Gra89].

In comparison to all these languages, the combination of multiple heads, propagation rules and logical variables with built-in constraints is unique for CHR.

2.5 Exercises

Exercises

2.1 Given some cities as propositional CHR constraints, write some propagation rules that can express which city can be reached directly from which other city. What is the answer to a query that consists of a city? How do you avoid nontermination?

Now use unary CHR constraints of the form `city(NameOfCity)`. What becomes simpler, what not? Use a set of CHR constraints `direct(City1,City2)` to implement the propagation rules.

Answer:

```
ulm \ ulm <=> true.
...
ulm ==> munich.
munich ==> ulm.
munich ==> salzburg.
...
```

All cities reachable form the city in the query are computed. The duplicate removal rule avoids nontermination.

```
city(A) \ city(A) <=> true.
...
city(ulm) ==> city(munich).
city(munich) ==> city(ulm).
city(munich) ==> city(salzburg).
...
```

Simpagation rules for each city can be merged into one. Propagation rules become more verbose.

```
city(A) \ city(A) <=> true.
...
direct(City1,City2), city(City1) ==> city(City2).
...
```

2.2 Represent colors as propositional CHR constraints `red, blue,....` Write simplification rules that describe the result of mixing two primary colors. Observe what happens if you have all three primary colors, in different orders, in the query. How to ensure that the answer is always the same, say `brown`?

Answer:

```
red, blue <=> violet.
red, yellow <=> orange.
blue, yellow <=> green.
red, blue, yellow <=> brown.
```

Confluence can be regained by additional rules, e.g.
`violet, yellow <=> brown.`

2.3 In an example from geometry, assume that lines are given by variables (or constants) and that CHR constraints express the relationships between two lines, `parallel` and `orthogonal`. Write propagation rules that derive further such relationships from the given relationships. Ensure termination.

Answer:

```
parallel(L1,L2) \ parallel(L1,L2) <=> true.
...
parallel(L1,L2), parallel(L2,L3) ==> parallel(L1,L3).
orthogonal(L1,L2), orthogonal(L2,L3) ==> parallel(L1,L3).
...
```

2.4 Compute the factorial of a number `n`, given `fact(1),...,fact(n)`.

Answer:

```
fact(N), fact(M) <=> fact(N*M).
```

2.5 Extend the Prime numbers program to factorize numbers.

2.6 What happens if the exchange sort rule is run backwards, i.e. head and body are exchanged?

2.7 The exchange sort rule can be restricted to considering only neighboring array entries.

```
a(I,V), a(J,W) <=> I=:=J+1, V<W | a(I,W), a(J,V).
```

By a similar reasoning as before we can still be sure that the final array will be sorted.

2.8 What does the exchange sort rule with array constraints where value and index are exchanged, i.e. `a(Value, Index)`?

2.9 Add several natural numbers. Write numbers in successor notation, i.e. `s(s(0))` denotes the number 2. Use two CHR constraints, `add` for each number to add, and `sum` for the resulting sum of all the numbers.

Our query then has the form `add(V1),...,add(Vn),sum(0)`. In the rules consider one `add` constraint at a time. If it contains zero, delete it, because it cannot change the sum. If it is not zero, it is the successor of some number `X`, and so move one successor symbol to the sum.

Answer:

```
add(0) <=> true.
add(s(X)), sum(Y) <=> add(X), sum(s(Y)).
```

The second rule applies until all `add` constraints contain zero, at which point they will be deleted and only the output remains with the sum.

2.10 Compute the number of days in a given year. The calculation is based on the modulo operation for the year and rules to find out, if the given year is a leap year. For example, consider the years 1991, 2000, 2004.

Answer:

```
days(Y,D) <=> days(Y mod 4, Y mod 100, Y mod 400, D).

days(Ym4, Ym100, 0, D) <=> D=366.
days(Ym4, 0, Ym400, D) <=> Ym400>0 | D=365.
days(0, Ym100, Ym400, D) <=> Ym100>0 | D=366.
days(Ym4, Ym100, Ym400, D) <=> Ym4>0, Ym400>0 | D=365.
```

2.11 In the Newton approximation of the square root, extend the constraint `improve` with a counter.

2.12 Like the Newton approximation of the square root, implement the approximation of the reciprocal using the formula $G_{i+1} = 2G_i - XG_i^2$.

2.13 Binomial coefficients (Pascal's triangle) can be computed based on the double recursive definition (in functional notation):
$cb(n,k) :=$
if $k = 0$ or $k = n$ then 1
if $k > 0$ and $k < n$ then $cb(n-1,k) + cb(n-1,k-1)$
where $0 \leq k \leq n$. Use a CHR constraint `cb(N,K,B)` and experiment

with top-down and bottom-up implementations in analogy to the Fibonacci example.

2.14 Consider the program for transitive closure. What happens if we replace `p(X,Y)` in the first propagation rule of the original program by `p(X,X)`? What happens in the program variations?

2.15 Consider the program for transitive closure. To ensure termination, we may have the idea to avoid applications of the second propagation rule when it produces some path again. The same path is produced again if `X=Y`. Similarly, if `Y=Z`, we will produce a path that can already be produced by the first propagation rule. Is this sufficient to ensure termination?

Answer: We add a guard to the original propagation rule:

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

However this alone is not enough to guarantee termination, as soon as we have more than one cycle in the graph. Consider `e(1,2), e(2,3), e(3,1), e(2,1)`.

2.16 Consider the program for computing path lengths. What happens if we replace the guard `N=<M` of the modified duplicate elimination rule by `N=M`?

Answer: For the query `e(X,Y), e(Y,Z), e(X,Z)` the answer will be `p(X,Z,1), p(X,Z,2), p(Y,Z,1), p(X,Y,1)`. The answer shows that there are two paths of different length from node `X` to node `Z`. The duplicate rule only removes paths that are also identical in the length. But this now causes nontermination, since e.g. already `e(X,X)` has infinitely many paths with different lengths from `X` to `X` itself (we essentially just count the path length up).

2.17 For a car routing application, where nodes correspond to cities, extend the edge constraint `e` by a third argument that contains the distance between the two cities. Adapt the shortest path program accordingly.

2.18 Consider the program for transitive closure, modified for single sources or target nodes. What happens if there are several target and/or source nodes?

2.19 Based on the for transitive closure, implement rules for the single-source shortest path computation.

Answer: Specialize the propagation rule with `source/1` and extend to shortest paths:

```
source(X),e(X,Y) ==> p(X,Y,1).
source(X),p(X,Y,N),e(Y,Z) ==> p(X,Z,N+1).
```

2.20 Find the shortest path from a given source vertex $s \in V$ to all other vertices $v \in V$ in a weighted directed graph $G = (V, E)$. The weight-function $w : E \to \mathbf{R}$ is lifted for a path $p = \langle v_0, v_1, ..., v_k \rangle$ to $w(p) = \sum_{i=1}^{k} w(v_{i-1}, v_i)$. The shortest path between vertices u and v is the minimum weight of all paths $u \mapsto v$, or if there is no such path it is ∞.

To avoid negative weight cycles we allow nonnegative weights only.

a) Write a CHR program to solve the SSSP problem using the standard relaxation method.

b) Enhance your program, s.t. for all vertices $v \in V$ the path from s to v yielding minimal weight is stored.

2.21 In a connected Eulerian graph, there exists a path that traverses all nodes. Write a simple program to check if a graph is Eulerian. It suffices to check if each node has the same number of incoming and outgoing edges.

Answer:

```
e(X,Y) ==> e_in(X,Y), e_out(X,Y).
e_in(X,Y), e_out(Y,Z) <=> true.
```

The graph is Eulerian, if no auxiliary edges `e_in` and `e_out` are left.

2.22 Take your last name with `leq` constraints from the partial order example program 2.4.2 between succeeding characters written as variables. For example, the name *Fruehwirth* translates to the query
`F leq R, R leq U, U leq E, E leq H, H leq W, W leq I, I leq R, R leq T, T leq H`
and leads to the answer
`F leq E, H=E, I=E, R=E, T=E, U=E, W=E.`

2.23 Define a grammar that admits repetitive sentences about roses, “a rose is a rose”, “a rose is a rose is a rose”,... where the terminal symbols are the words in such a sentence.

2.24 Write a rule for admitting grammar rules of the form `A->empty`, where the special symbol `empty` stands for the empty string. These type of rules is usually allowed for regular languages. This extension does not change the expressitivity.

2.25 What happens to the merge sort rule if the guard is weakened to allow the rule to be applicable to `A->A` arcs?

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

Answer: If the rule guard allows `A=B`,

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

then the query `A -> A, A -> C` removes and adds `A -> C` again and again. The query does not terminate.

2.26 Consider the classical *Hamming's Problem*, which is to compute an ordered ascending chain of all numbers whose only prime factors are 2, 3 or 5. The chain starts with the numbers
$(1, 2, 3, 4, 5, 6, 8, 9, 10, 12, 15, 16, 18, 20, 24, 25, \ldots$.
Two neighboring numbers later in the chain are 79164837199872 and 79254226206720.

Generate the infinite sequence of Hamming numbers using the merge sort rule with duplicate removal.

The idea for solving this problem is based on the observation: any element of the chain can be obtained by multiplying a previous number of the chain with 2, 3 or 5. The only exception is the initial number 1.

Define a nonterminating process `hamming(N)` that will produce the numbers as elements of the infinite chain starting with value `N`. We multiply the number `N` with 2, 3, and 5, and merge them using merge sort. Once we have done this, we know that the successor of `N` in the chain must be determined, and we can move along the arc starting in `N` to recursively call `hamming` with that new value.

Answer: To the rules for merge sort we add one of the following pair of rules:

```
hamming(X) <=> X->X*2, X->X*3, X->X*5, next(X).
X->A \ next(X) <=> writeln(X), hamming(A).

hamming <=> start->1, next(1).
X->A \ next(X) <=> writeln(X), A->A*2, A->A*3, A->A*5, next(A).
```