Rule-based Programming

Prof. Dr. Thom Frühwirth, Daniel Gall

Summer Term 2014

Assignment #1

Introducing Constraint Handling Rules (CHR)

Exercise 1. Installation

(1) Download and install SWI-Prolog `http://goo.gl/RSj8i` to your computer. It is also installed in all PC pools.
(2) If you are using Windows, you can then download and install the SWI-Prolog Editor: `http://goo.gl/6deJX`
(3) Use the SWI-Prolog manual on how to use the CHR library: `http://goo.gl/wNvQw`

Exercise 2. Hello World

(1) Some basic coding rules:
   (a) All code lines must end with a dot.
   (b) Constraint names and values must begin with a small letter.
   (c) Variables (unknowns) must begin with capital letters.
   (d) Constraint names, variables and values cannot have spaces.
   (e) To add comments which are not be executed, use `%` followed by the comment.
(2) Before using CHR rules, the CHR library must be included by:
```
:- use_module(library(chr)).
```
(3) User-defined constraints must be declared with their name and arity (number of arguments), as:
```
:- chr_constraint  name/arity.
```
(4) Write a "Hello world!" program in CHR:
```
:- use_module(library(chr)).
:- chr_constraint start/0.
start <=> write('Hello world!').
```
   Run the program with the query: `start.`
(5) Use `chr_trace` to switch on the tracer and view interactions with the constraint store.
(6) End the tracing using `chr_notrace`.

Exercise 3 (Walking by Simplification)**.** A walk can be expressed by movements `east, west, south, north`. The number of steps required can be simplified by the following rules:

```
east, west <=> true.
south, north <=> true.
```

Implement the walking rules in CHR, and test it for queries like: '`east, south, west, west, south, south, north, east, east`'.

Exercise 4 (Carry Less)**.** A coin exchange machine aims to help people carry less coins. It works by reducing the number of coins that a person would carry. For example, instead of carrying four 50-cents, the machine replaces them with one 2-euro coin. Assume that the least acceptable coin is the 10-cent coin. Model the coins using the constraints `euro2, euro1, cent50, cent20, cent10`, and then write a minimal number of CHR rules that would collect the smaller-valued coins to produce less bigger-valued ones. Test your program with the queries below:

(1) `cent10,cent10,cent10,cent10,cent10` → `cent50`
(2) `cent20,cent20,cent10,cent10,cent10,cent10,cent10,cent10` → `euro1`
(3) `euro1,cent50,cent50` → `euro2`
(4) `euro1,cent20,cent20,cent20,cent20,cent20,cent50` → `euro2,cent50`

Exercise 5 (Ro-Sham-Bo)**.** Rock-paper-scissors or "Ro-Sham-Bo" is a hand game usually played by two people, where players simultaneously form one of three shapes with an outstretched hand. The `rock` beats `scissors`, the `scissors` beat `paper` and the `paper` beats `rock`; if both players throw the same shape, the game is tied (Wikipedia). Write some CHR rules that determine the winner of a round of this game or detects a draw.

Exercise 6 (Age Calculator by Propogation)**.** Constraints can be used like a database to store and process information about famous actors. The date of birth of a person can be expressed using a `dob/2` constraint, like `dob(tom-cruise, 1962)`. The current year can be expressed as `today(2014)`. Write a CHR rule that preserves knowledge already stored, and calculates the current age of an actor, storing it in an `age/2` constraint. (Assume that the exact day and month do not matter). Your program is correct if it reveals that Tom Cruise is 52 years old!

Exercise 7 (Minimum by Simpagation)**.** Many numbers can be given as a query `min(`n_1`),` `min(`n_2`),...,` `min(`n_k`)`. Write a single CHR rule that filters these numbers, such that the remaining one is the least (or minimum) one.

The next exercises are to be submitted by e-mail to: daniel.gall@uni-ulm.de. The deadline is on 07.05.14 by 12:00. You are allowed to work in a group of two people. Please send only one e-mail per group, containing the solution and both team member names.

Exercise 8 (Water)**.** Water molecules can be produced from hydrogen and oxygen molecules if they are heated. With electricity, the water molecules get decomposed into hydrogen and oxygen molecules. These chemical reactions can be expressed as:

$$\text{heat} + 2H_2 + O_2 \rightarrow 2H_2O$$
$$\text{electricity} + 2H_2O \rightarrow 2H_2 + O_2$$

Using CHR constraints `h2, o2, h2o, heat, electricity` and assuming that one heat or electricity unit is needed for each reaction, write CHR rules to model these reactions. Test your program with queries like:

- `heat,h2,h2,o2` → `h2o, h2o`
- `heat,h2,h2,o2,h2,h2,o2` → `h2,h2,o2,h2o,h2o`
- `heat,h2,h2,o2,h2,h2,o2,heat` → `h2o,h2o,h2o,h2o`
- `heat,h2,h2,o2,h2,h2,o2,heat,electricity` → `h2,h2,o2,h2o,h2o`
- `electricity,electricity,h2o,h2o,h2o,h2o` → `h2,h2,h2,h2,o2,o2`

Exercise 9 (Better Age Calculator)**.** Modify the age calculator program so that it calculates the exact age of a person. Dates can be expressed by 3 arguments: the day, the month and the year. The current date can be given as `today(29,4,2014)`. Moreover, the date of birth of a person is now represented as a `dob/4` constraint to also include the month and day. You will need to define multiple rules to calculate the exact age, with different guard expressions.

Your corrected program reveals that Tom Cruise is actually only 51 years old!

Hint: Test your program with more examples to make sure that you cover all possible cases.

Rule-based Programming

Prof. Dr. Thom Frühwirth, Daniel Gall

Summer Term 2014

Assignment #2

Refer to the SWI-Prolog reference manual `http://www.swi-prolog.org/pldoc/refman/` *for documentation on the usage of the built-in predicates and the CHR library.*

Exercise 1 (Online CHR)**.** Add the following two lines to any CHR program to make it more interactive. It allows to add more constraints incrementally and then shows the resultant constraint store. To invoke this interactive run, add `more` at the end of your initial query. Then type a semi-colon `;` to provide more input.

```
:- chr_constraint more/0.
more <=> true ; (read(Constraint), call(Constraint), more).
```

Test this with any of the examples from assignment #1.

In the next exercises we are going to implement a (file-)writer for CHR rules. The tool will be used in the next assignments when we transform programs of other rule-based formalisms to CHR.

Exercise 2 (CHR-Writer)**.** A generalized CHR simpagation rule which is given as follows:

```
Head1 \ Head2 <=> Guard | Body.
```

can be represented using a `chrl/4` constraint in the form:

```
chrl(Head1, Head2, Guard, Body)
```

where `Head1, Head2, Guard, Body` are lists of CHR and built-in constraints. If they do not exist (i.e. for propagation or simplification rules, or guard-less rules), then they are given as `[]`. Implement a CHR program that once triggered with a `console/0` constraint, writes to the console a well-formatted CHR rule equivalent to that encoded in `chrl` constraint by changing the lists to goals.
Test your program with (but not limited to) queries such as:

- `chrl([],[a,b],[],[c])` represents: `a,b <=> true | c.`
- `chrl([a,b],[],[],[c])` represents: `a,b ==> true | c.`
- `chrl([a],[b],[],[c])` represents: `a \ b <=> true | c.`
- `chrl([a(X)],[b(Y,Z)],[X<Y],[c(Z)])` represents: `a(X) \ b(Y,Z) <=> X<Y | c(Z).`

Hints:

- For each `chrl/4` constraint, the lists must be transformed to goals. Write helper rules for the constraint `list2goal(L,G)` which transfers a list of atoms (e.g. `[a,b,c]`) to a goal `a,b,c`. Concatenation of two goals can be achieved by `(G1,G2)`.
- Save the goals in a `chr/4` constraint. Each of those constraints is then transformed to the output of the corresponding CHR rule.
- Use `numbervars` to pretty print a term by unifying variables ($\_123, \_456, \ldots$) to more readable variable names ($A, B, \ldots$).

Exercise 3 (CHR-File-Writer)**.** Modify the program of the previous exercise; such that instead of displaying on the console, the output CHR rule is written to a file once triggered by a `file/1` constraint. *Hint:* Use built-in `open/3`, `write/2` and `close/1` to write to file streams. A file stream must be closed after writing to. You may find it useful to have rules for the constraints: `file(new), file(append), file(close).`

Exercise 4 (File-Reader)**.** Implement a program that reads a text file when triggered with a `file(read)` constraint. It should transform each line into a `line/1` constraint. *Hint:* Use `read/2` to read a term from a stream.

Next exercises are to be submitted by e-mail to: daniel.gall@uni-ulm.de by 14.05.14 by 10:00.

Exercise 5 (Hobo Cigarettes)**.** A certain hobo can make one cigarette out of four cigarette butts (the butt is what is left after smoking a cigarette). If he finds some cigarettes and some cigarette butts, how many cigarettes can he smoke in total?
The input is any number of constraints of this form: `cigarette/0`, `butt/0`, and `pack/1`. Each cigarette represents one cigarette, and each butt represents one cigarette butt. The constraint `pack(N)` represents a pack containing `N` cigarettes. The output is one constraint of the form `smoked(T)`, where T is the total number of cigarettes the hobo has smoked. There can be no cigarettes left (the hobo smokes every cigarette he finds or makes), but there can be (will be) cigarette butts left (always less than four). Examples:

```
?- pack(4).
butt
smoked(5)
```

```
?- pack(25).
butt
smoked(33)
```

```
?- butt, pack(3).
butt
smoked(4)
```

Exercise 6 (Exchange Sort)**.** An array can be represented as a multiset of pairs of the form `a(Index,Value)`. Implement the exchange sort algorithm that sorts the array of numbers by exchanging values at positions that are in the wrong order. Test it with an appropriate query.

Exercise 7 (Hamming's problem)**.** 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,...`

Implement the problem in your favourite programming language. Within your submission-group, prepare 2-3 presentations slides to present the idea of your solution. You will present your solution to the entire class for discussion on 15.05.14.

Rule-based Programming

Prof. Dr. Thom Frühwirth, Daniel Gall

Summer Term 2014

Assignment #3

Term Rewriting Systems (TRS)

Exercise 1 (Flattening)**.** Define `eq` to be a binary CHR constraint in infix notation denoting equality, using `op/3`. Write a CHR rule that implements the flattening function that transforms an atomic equality constraint `X eq T`, where `X` is a variable and `T` is a term, into a conjunction of equations as follows, where $X_1, \ldots, X_n$ are new variables.:

$$\texttt{flatten}(X\ \texttt{eq}\ T) := \begin{cases} X\ \texttt{eq}\ T & \text{if } T \text{ is a variable} \\ X\ \texttt{eq}\ f(X_1, \ldots, X_n)\ \wedge\ (\bigwedge_{i=1}^{n}[X_i\ \texttt{eq}\ T_i]) & \text{if } T\ =\ f(T_1, \ldots, T_n) \end{cases}$$

Hint: You can change terms to list of functor and arguments by: `f(X1,...,XN)=..[f,X1,...,XN]`

Implement a second version of the flattening rule, implemented by having a `flatten/2` which flattens a term passed as the first argument into a list (as the second argument).

Exercise 2 (Translate TRS to CHR)**.** It is required to translate TRS rules into CHR simplification rules. A CHR constraint `translate/2` has the type `trs` as the first argument and the second containing a TRS rule of the form:

$$S\ \texttt{-->}\ T$$

Write a rule that transforms the encoded TRS rule to a CHR simplification rule (with X as a new variable):

$$\texttt{flatten}(X\ \texttt{eq}\ S)\ \texttt{<=>}\ \texttt{flatten}(X\ \texttt{eq}\ T)$$

The output CHR rule can be encoded in a `chrl/4` constraint as in assignment #2, thus as:

$$\texttt{chrl([],[flatten}(X\ \texttt{eq}\ S)\texttt{],[],[flatten}(X\ \texttt{eq}\ T)\texttt{])}$$

The translator rule invokes the CHR-writer of assignment#3 to output the CHR simplification rules to the console or file. *Hint:* You will need to define the binary operator `(-->)`. Test your translator with appropriate examples.

Enhance your program with an additional constraint capable of reading an input text file containing TRS rules and producing a translated CHR program written in an output text file.

Exercise 3 (Translate to CHR)**.** Two rewrite rules that define the addition of natural numbers in successor notation, are:

```
0+Y --> Y.
s(X) + Y --> s(X+Y).
```

Translate the rules into CHR using by applying the flattening function manually. Then use your translator to show its output result. Include in your program 6 appropriate test examples to show the correctness of your work; these examples can be present as comments.

Exercise 4 (Propositional Logic)**.** Given the following TRS for conjunction in propositional logic, where `X`, `Y` and `Z` are propositional variables and the function `and(X,Y)` stands for `X` $\wedge$ `Y`:

```
and(0,Y) --> 0.
and(X,0) --> 0.
and(1,Y) --> Y.
and(X,1) --> X.
and(X,X) --> X.
```

Write down similar TRS rules for negation, `neg`, and disjunction, `or`, in propositional logic. Run the translator and produce the equivalent CHR rules for conjunction, disjunction and negation. Include in your program 6 appropriate test examples to show the correctness of your work; these examples can be present as comments.

Exercise 5 (Run Translated TRS Program)**.** To run a TRS program in CHR, the query must be flattened first. Write a rule for the constraint `evaluate(Query)` which flattens the `Query` then triggers the TRS evaluation. Test the correctness of the TRS programs by evaluating some test queries.

Sample run for the evaluation of the number addition translated program:

```
?- evaluate(X eq s(s(s(0)))+s(0)).
X = s(s(s(s(0))))
```

Exercise 6 (Functional Dependency)**.** Augment your program with a functional dependency simpagation rule that implements structure sharing to ensure completeness. This CHR rule must be come first within the program.

```
fd @ X eq T \ Y eq T <=> X=Y.
```

The `fd` rule removes equations, thus some rules may not be applicable anymore. For example, for the TRS rule:

```
and(X,X) --> X
```

which translates in the CHR rule:

```
T eq and(T1,T2), T1 eq X, T2 eq X <=> T eq X.
```

The rule expects two copies of the equations `T1 eq X` and `T2 eq Y`. Thus variants of the existing CHR rules must be created, where head constraints have been unified such that the rules apply after the `fd` rule has fired. For the previous example, this results in the additional rule:

```
T eq and(T1,T1), T1 eq X <=> T eq X.
```

Manually write down all other additionally required rules for the other conjunction and disjunction rules.

Next exercise is to be submitted by e-mail to: daniel.gall@uni-ulm.de by 21.05.14 by 10:00.

Exercise 7 (Structure Sharing, Bonus)**.** Write a CHR program which produces automatically the extra rules generated by structure sharing of exercise 6 for any TRS rule during translation.

Exercise 8 (Shortest Paths)**.** The following program takes a directed graph, where the edges are represented as `e/2` constraints (`e(A,B)` means that there is a (directed) edge from `A` to `B`), and computes the reachability relation `p/2` (where `p(A,B)` means that there is some path from `A` to `B`).

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

Modify the above program such that it computes the distance relation `d/3`, where `d(A,B,N)` means that the shortest path to go from `A` to `B` uses `N` edges. (Try for queries like `e(a,b), e(a,c), e(b,c), e(b,d), e(d,f)`)

Rule-based Programming

Prof. Dr. Thom Frühwirth, Daniel Gall

Summer Term 2014

Assignment #4

Functional Programming (FP)

Exercise 1 (Translating FP to CHR)**.** It is required to translate FP rewrite rules into CHR simplification rules. A CHR constraint `translate/2` has the type `fp` as the first argument and the second containing an FP rule of the form:

$$S \texttt{ --> } (G) \texttt{ | } T$$

Write a CHR rule which translates it to a CHR simplification rule:

$$X \texttt{ eq } S \texttt{ <=> } G \texttt{ | flatten(} X \texttt{ eq } T\texttt{)}$$

where X is a new variable. The rule should write the output CHR simplification rules to the console, by translating the FP rule to a `chrl/4` constraint and triggering the writer from assignment #2. Test your translator with appropriate examples. *Hint:* You will need to define the binary operator `(-->)` . Also consider that some FP rules can be written without a guard, i.e. as S `-->` T.

Enhance your program with an additional constraint capable of reading an input text file containing FP rules and producing a translated CHR program written in an output text file.

Exercise 2 (Executing FP in CHR)**.** The CHR program produced by the translator implemented above, requires additional rules for treating data and mapping auxiliary functions to built-in constraints. It should be able to evaluate built-in contraints and bind ground values to variables. Using the built-in SWI predicates: `arithmetic_expression_value/2, number/1, var/1, ground/1`, write the appropriate CHR rules that are needed for the execution of any translated FP program.

Then, modify your translator such that it augments the additional rules to the translated FP rules written in the beginning of the output file. It should also add the required CHR header lines such that the output code is executable.

Exercise 3 (Fibonacci Numbers)**.** The sequence of Fibonacci numbers is defined as follows:

$$F_0 = 0, F_1 = 1$$
$$F_n = F_{n-1} + F_{n-2}, for\ n \geq 2$$

Write the FP rules that calculate the Fibonacci of a number. Translate the rules into CHR manually. Then use your translator using the input FP rules to show its output CHR program. Try your modified translator to produce the CHR code for the translation of the Fibonacci code. Then try to run the Fibonacci code and test is for obtaining several of the Fibonacci numbers.

Next exercises are to be submitted by e-mail to: daniel.gall@uni-ulm.de by 28.05.14 by 10:00.

Exercise 4 (Factorial)**.** The factorial of a non-negative integer n is the product of all positive integers less than or equal to n. Write the FP rules for the factorial problem. Produce the equivalent CHR code by the translator, and run it with appropriate test queries.

Exercise 5 (Translating CHR to FP)**.** Go back through the first examples presented in earlier lecture slides. Which of these CHR examples can be expressed in Functional Programming? Also present their equivalent code in Functional Programming. Which examples cannot be expressed, and why?

Rule-based Programming

Prof. Dr. Thom Frühwirth, Daniel Gall

Summer Term 2014

Assignment #5

Functional Programming: Higher Order Functions

Higher order functions are functions which take at least one function as input. For example, the function

```
twice(F,A) --> F(F(A)).
```

applies the function `F` to argument `A` twice.
In this lab session, we want to integrate such functions to our functional programming language. Since `F(A1,...,An)` is not a valid Prolog term, we require for the syntax of our functional language that such applications are expressed by a term `apply(F,[A1,...,An])` denoting that function `F` is applied to the arguments `A1` to `An`.

Exercise 1 (Apply)**.** Write a CHR rule which handles the function application. I.e. write a rule which rewrites a term `apply(F,[A1,...,An])` to `F(A1,...,An)` if `F` is ground.

Usually, functional programming languages provide some default functions including some higher order functions. In the following, we want to implement some well-known higher order functions using our functional language and analyze the results of our translator.

Exercise 2 (Map)**.** Write a function `map(F,L)` which applies the function `F` to each element in the list `L` (returning a list). Test your implementation with some examples. Which examples do not work out of the box?

Exercise 3 (Fold)**.** Make yourself familiar with the concept of the `foldr(F,E,L)` function and implement it in our functional language. Test your implementation with some examples (e.g.: Write test calls of `foldr` to calculate the sum and the product of a list of numbers).

Next exercises are to be submitted by e-mail to: daniel.gall@uni-ulm.de by 04.06.14 by 10:00.

Exercise 4 (Faculty)**.** Redefine your faculty function from assignment #4 using foldr.
Hint: Write a function `make_list(N)` which produces a list containing all numbers from `1` to `N` (without respecting order). Make sure you use guards to check that `N` is a number! Then use the product example from exercise 3 on this list.

Rule-based Programming

Prof. Dr. Thom Frühwirth, Daniel Gall

Summer Term 2014

Assignment #6

GAMMA

Exercise 1 (Translating GAMMA to CHR)**.** It is required to translate each GAMMA pair into a CHR simplification rule. Implement a rule for the CHR constraint `translate/2` that transforms a GAMMA pair:

$$(c/n, f/n)$$

into a CHR simplification rule:

$$d(x_1), \ldots, d(x_n) \texttt{ <=> } c(x_1, \ldots, x_n) \texttt{ | } f(x_1, \ldots, x_n)$$

where `d/1` is a CHR constraint that wraps the data elements, and `c/1` is a built-in constraint that checks for a certain condition. The function f is manually defined by a simplification rule of the form:

$$f(x_1, \ldots, x_n) \texttt{ <=> } G \texttt{ | } D,\ d(y_1), \ldots, d(y_m).$$

where G is a guard and D are the auxiliary built-ins. The built-in constraint `c/n` is a Prolog test predicate, that can be manually defined explicitly per problem. Assume that the definitions for `f/n` and `c/n` will be done separately.

Exercise 2 (GAMMA Lecture Examples)**.** Test your translator with the following GAMMA examples from the lecture slides:

```
min = (</2, first/2)
gcd = (gcd_check/2, gcd_sub/2)
prime = (div/2, first/2)
```

For each example, define the CHR rule for the function `f/2` and the Prolog predicate for `c/2`, also give the translated CHR rule. Test the produced CHR codes with appropriate queries.

Exercise 3 (Translating CHR to GAMMA - Mergers and Acquisitions)**.** A large company will buy any smaller company. A CHR constraint `company(Name,Value)` can be defined where `Value` is the market value of the company. A rule that describes the merge-acquisition cycle that is observed in the real world is given as:

```
company(Name1, Value1), company(Name2, Value2)
        <=> Value1 > Value2 | company(Name1:Name2,Value1+Value2).
```

Translate the Company Mergers CHR program into GAMMA, stating the CHR rule for the function `f` and the Prolog predicate for `c`. Run the translator on the GAMMA pair, and show that the output CHR rules are semantically equivalent to the initial CHR program.

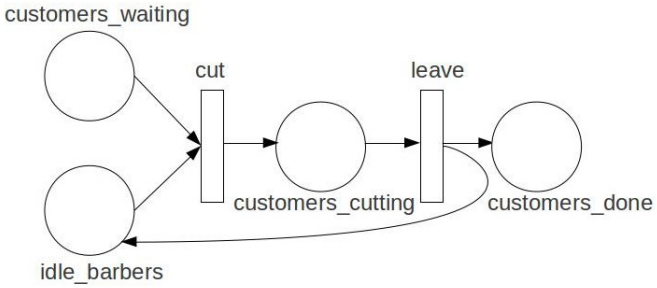
Exercise 4 (Translating CHR to GAMMA - Walk)**.** Assume we describe a walk (a sequence of steps) by giving directions, `east, west, south, north`. A description of a walk is just a sequence of these CHR constraints. With simplification rules, we can model the fact that certain steps (like `east, west`) cancel each other out, and thus we can simplify a given walk to one with a minimal number of steps that reaches the same position.

```
east, west <=> true.
south, north <=> true.
```

Translate the Walk CHR program into GAMMA, stating the CHR rule for the function `f` and the Prolog predicate for `c`. Run the translator on the GAMMA, and show that the output CHR rules are equivalent to the initial CHR program. Test the produced CHR codes with appropriate queries.

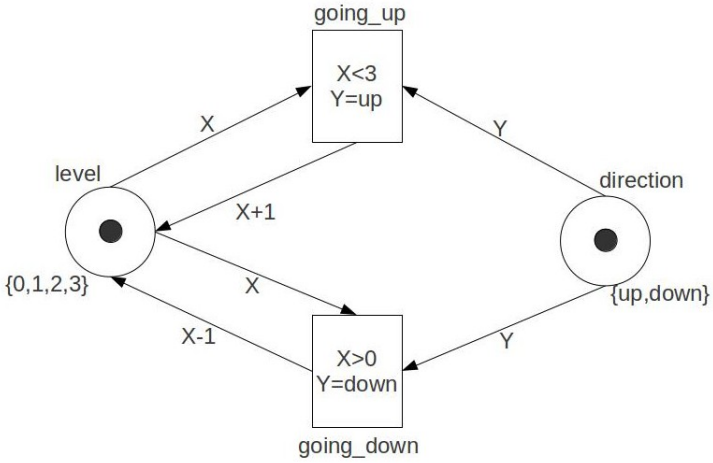
Petri Nets

Exercise 5 (Petri Nets - Barber Shop)**.** A typical scenario at a Barber shop is as follows: Customers enter a Barber shop and wait till a barber is idle and ready to serve them. Then the barber cuts the hair of a customer. When the hair cut is done, the customer leaves and the barber becomes idle once again. This can be represented using the Petri net given below:



Translate the Barber shop Petri net into CHR by adding the constraints `customers_waiting/0`, `idle_barbers/0`, `customers_cutting/0`, and `customers_done/0` for each of the places. Add an `observer/0` constraint to print the interesting states of the problem. A typical test query would be `?-observer,customers_waiting,customers_waiting,idle_barbers.`

Exercise 6 (Colored Petri Nets - Elevator)**.** An elevator operates between 4 levels, a petri-net is designed such that a level token represents the level of the elevator and the direction tokens represent the required movements of the elevator. The elevator can go upwards, increasing its level only if it is not on the maximum floor. Similarly the elevator can go down, decreasing its level only if it is not on the ground floor. The system can be represented using the following colored Petri net:



Translate the elevator Petri net into CHR by adding the constraints `level/1, direction/1` and the equivalent transition rules. Add an `observer/0` constraint to print the interesting states of the problem. A typical test query would be
`?- observer, level(0), direction(up),direction(up),direction(up),direction(down).`

Next exercise is to be submitted by e-mail to: daniel.gall@uni-ulm.de by 11.06.14 at 10:00.

Exercise 7 (Translating CHR to GAMMA - More examples)**.** For each of the following problems, write down the CHR rules to solve them. Then translate the CHR program into GAMMA, stating the CHR rule for the function `f` and the Prolog predicate for `c`. Run the translator on the GAMMA pair, and show that the output CHR rules are semantically equivalent to the initial CHR program. Test the produced CHR codes with appropriate queries.

(1) **Exclusive OR** - A multi-set of `xor` constraints denoting the output can be used to compute the output as a single remaining `xor` constraint where truth values `true` and `false` are represented by the numbers `1` and `0` respectively.

(2) **Exchange Sort** - Sort an array by exchanging values at positions that are in the wrong order, given an unsorted array as a sequence of constraints of the form `a(Index,Value)`.

(3) **Destructive Assignment** - In a declarative programming language, bound variables cannot be updated or overwritten. However, in CHR it is possible to simulate the destructive (multiple) assignment of procedural and imperative programming languages by using recursion in CHR. The original constraint with the old value is removed and a constraint of the same type with the new value is added. For example, we can store variable name-value pairs in the CHR constraint `cell/2` and use the CHR constraint `assign/2` to assign to a variable a new value.

(4) **Merge sort** - To represent a directed edge (arc) from node A to node B, we use a binary CHR constraint written in infix notation, `A -> B`. We use a chain of such arcs to represent a sequence of values that are stored in the nodes, e.g. the sequence `0,2,5` is encoded as `0 -> 2, 2 -> 5`. A one-rule CHR program performs an ordered merge of two chains by zipping them together, provided they start with the same (smallest) node.

Rule-based Programming
Prof. Dr. Thom Frühwirth, Daniel Gall
Summer Term 2014

Assignment #7

Production Rule Systems - OPS5

An OPS5 production rule:

```
(p N LHS --> RHS)
```

translates to the CHR generalized simpagation rule:

```
N @ LHS1 \ LHS2 <=> LHS3 | RHS'
```

where

- `LHS`: if-clause
- `RHS`: then-clause
- `LHS1`: patterns of `LHS` for facts not modified in `RHS`
- `LHS2`: patterns of `LHS` for facts modified in `RHS`
- `LHS3`: conditions of `LHS`
- `RHS'`: `RHS` without removal (for `LHS2` facts).

Exercise 1 (Fibonacci)**.** The OPS5 rule for the iterative Fibonacci sequence generation is given as:

```
(p next-fib (limit ^is <limit>)
   {(fibonacci ^index {<i> <= <limit>}
               ^this-value <v1>
               ^last-value <v2>) <fib>}
   --> (modify <fib> ^index (compute <i> + 1)
                     ^this-value (compute <v1> + <v2>)
                     ^last-value <v1>)
       (write (crlf) fib <i> is <v1>)
)
```

Translate the OPS5 program to CHR by hand. Check your result with the Online CHR Translator (http://pmx.informatik.uni-ulm.de/chr/translator/).

CHR Rules with Negation as Absence

Exercise 2 (CHR with Negation)**.** An extension of CHR deals with rules which fire if a certain condition is negated or when a constraint does not exist in the store (i.e. negation as absence). These rules can be expressed as:

```
N @ Hk \ Hr <=> G | B : (NegB, NegC).
```

Using an auxiliary `check/1` constraint, it translates to the following CHR rules:

```
N1 @ Hk, Hr ==> G | check(Hk,Hr).
N2 @ NegC \ check(Hk,Hr) <=> NegB | true.
N3 @ Hk \ Hr, check(Hk,Hr) <=> G | B.
```

Write a rule for a `translate/2` constraint that takes a list containing a negated CHR rule and transforms it to an equivalent CHR program.

Exercise 3 (Negated Examples)**.** The minimum program can be written with negation as follows:

```
num(X) ==> min(X) : (Y<X,num(Y)).
```

Translate the rule to an equivalent CHR one and check with appropriate examples.

Write negated CHR rules for the transitive closure and marital status examples covered in the lecture. Translate them to normal CHR, and test with appropriate examples.

Exercise 4 (Special Case of CHR with Negation)**.** Negated CHR propagation rules whose body consists of only CHR constraints can be considered as a special case. For their transformation, it is not necessary to use the auxiliary `check` constraint.
These rules are of the form:

```
N @ Hk ==> G | Bc : (NegB, NegC).
```

This translates to the following pair of CHR rules:

```
Nn @ NegC \ Bc <=> NegB | true.
Np @ Hk ==> G | B.
```

Write a rule for a `translate/2` constraint that takes a list containing a negated CHR rule and transforms it to an equivalent CHR program.

Exercise 5 (Special Negated Examples)**.** Rewrite the three programs of exercise 3 into the special case form and produce their transformed CHR programs. Test the resultant programs with appropriate examples.

Rule-based Programming

Prof. Dr. Thom Frühwirth, Daniel Gall

Summer Term 2014

Assignment #8

Incremental Conflict Resolution in CHR

Exercise 1 (*Step 1: Translation).* The definition for CHR rules is extended to generalized CHR simpagation rules (with a property P) which is given as follows:

```
Hk \ Hr <=> Guard | Body : P.
```

Hence the extended rule can be represented using a `chrl/5` constraint, where `Hk, Hr, Guard, Body` are lists:

```
chrl(Hk, Hr, Guard, Body, P)
```

In the lecture, a translation scheme was discussed which translates the rule into two other conflict resolution rules by introducing a `conflictset/1` constraint to gather rule bodies and then execute the chosen rule from the conflict set. Rule bodies can be represented as `rule(P,Hk,Hr,Body)`. Write a rule for a `translate/2` constraint that performs this transformation. Test with the examples from the lecture.

Exercise 2 (*Step 2: Additional Conflict Resolution Rules).* Which rules must be added to the output program for the conflict resolution?
Bonus: Modify your translator such that it adds the rules for the conflict resolution automatically to the output file.

Exercise 3 (*Step 3: Additional Choice Rules).* For the rule choice, the `choose/3` constraint selects a particular rule from the conflict set of rules depending on the property P stated in the initial translated CHR rule.

(1) $P = bfs$: selects a rule from the set to ensure the breadth first traversal of rules
(2) $P = random$: randomly selects a rule from the conflict set
(3) $P = N, number(N)$: selects the rule with the highest priority (P)
(4) $neg(C, G)$: negation as absence; the rule is applied if there are no CHR constraints `C` for which the guard `G` holds

Add three rules for cases 1 to 3 to the output file, simplifying the `choose/3` constraint such that the first argument contains the list of rule terms (whose first term specifies the choice criterion), the second argument is bound to the selected rule, and the third argument is bound to the remaining list. *(Please note that case 4 will be covered next week).*

Exercise 4 (*Example: Dijkstra).* Dijkstra's shortest path algorithm can be expressed by giving priority to the application of the CHR rules. A lower weight is given to shorter paths, while constructing a longer path has more weight and hence its rule would have lower priority. This can be encoded using the following rules:

```
d2 @ dist(X,N) \ dist(X,M) <=> N<M | true : 1.
dn @ dist(X,N), edge(X,Y,M) ==> P is N+2 | Z is N+M, dist(Y,Z) : P.
```

A typical test query would be:

```
?- edge(a,b,10),edge(a,c,2),edge(b,c,1),
   edge(b,a,10),edge(c,a,2),edge(c,b,1),dist(a,0),fire.
```

Perform step 1, by encoding the rules into `chrl/5` constraints. Then run the translator from step 1 to obtain the output CHR rules, augment them to the rules from steps 2 and 3. Test your code with appropriate queries and show the results.

Next exercise is to be submitted by e-mail to: daniel.gall@uni-ulm.de by 02.07.14 at 10:00.

Exercise 5 (*Example: Random Dice)*. A dice is thrown and the result can be a 1, 2, 3, 4, 5 or 6. The result is required to be random, thus the random execution of the rules can be encoded using the following set of CHR rules:

```
dice <=> write(1) : random.
dice <=> write(2) : random.
dice <=> write(3) : random.
dice <=> write(4) : random.
dice <=> write(5) : random.
dice <=> write(6) : random.
```

A typical test query and its results would be:

```
?- dice,fire.
   5

?- dice,fire.
   2
```

Perform step 1, by encoding the rules into `chrl/5` constraints. Then run the translator from step 1 to obtain the 12 output CHR rules. Create a new file, add the output 12 rules in addition to the rules from steps 2 and 3. Test your code with appropriate queries, and show the multiple random results.

Exercise 6 (*Example: Multiple Count-ups)*. The following CHR program which when given a `count/1` constraint, can either display it on the console or calculate the next count. It is required to perform a breadth-first-traversal of the rules. The CHR code is given as following:

```
count(X) ==> write(X) : bfs.
count(X) <=> Y is X+1, count(Y) : bfs.
```

A typical test query and its results would be (but with the output numbers on separate lines):

```
?- count(0),count(100),fire.
   0  100  1  101  2  102  3  103  4  104  5  105  6  106 ...
```

Perform step 1, by encoding the rules into `chrl/5` constraints. Then run the translator from step 1 to obtain the output CHR rules, augment them to the rules from steps 2 and 3. Test your code with appropriate queries, and show the multiple random results.

Rule-based Programming

Prof. Dr. Thom Frühwirth, Daniel Gall

Summer Term 2014

Assignment #9

Negation as Absence

Exercise 1 (*Case 4 of Step 3: Negation Choice Rule)*. As a continuation of Exercise#3 in Assignment#8, it remains to handle negation as absence. In CHR, a rule which checks for the absence of particular constraints is expressed as follows:

```
Heads1 \ Heads2 <=> Guard | Body : neg(NH,NG).
```

Write a rule for the 4th case such that the `choose/3` constraint when given an input list where the rule has `P = neg(NH,NG)` then: if the constraints `NH` are present and `NG` holds, then it should remove the rule with this negation and continue resolving the conflict, otherwise the rule with negation is chosen as the rule to apply. Hence, the translation itself should be changed. Add a rule to perform this modified translation.

Exercise 2 (*Example: Martial Status)*. A person is single or married, where single is to be the default. This can be expressed by a negated CHR rule as follows:

```
person(X) ==> single(X) : neg(married(X),true).
```

A typical test query and its results would be:

```
?- married(a), person(b).
married(a)    person(b)    single(b)
```

Encode the rule into an equivalent `chr/5` constraint, translate it then test the resultant program.

Exercise 3 (Example: Minimum as Negation). The minimum number amongst a multi-set of numbers can be expressed using the rule:

```
num(X) ==> min(X) : neg(num(Y), Y<X).
```

A typical test query and its results would be:

```
?- num(2), num(1), num(10).
num(10)    num(1)    num(2)    min(1)
```

Encode the rule into an equivalent `chr/5` constraint, translate it then test the resultant program.

Exercise 4 (Example: Graphs Closure). When finding the transitive closure of a graph, then a negated CHR program can be given as:

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

A typical test query and its results would be:

```
?- e(a,b), e(b,c), e(c,d).
e(c,d)  e(b,c)  e(a,b)  p(a,d)  p(b,d)  p(a,c)  p(c,d)  p(b,c)  p(a,b)
```

Encode the rule into an equivalent `chr/5` constraint, translate it then test the resultant program.

Priority Learning

In the following exercises, we want to simulate an intelligent agent who plays the game *rock, paper, scissors*. The player is confronted with several opponents with different strategies and preferences. For instance, one player might prefer to play rock in every move. Our player should learn the preferences of an opponent from his decisions. Therefore, we use techniques from computational cognitive modeling to implement a priority learning in our conflict resolution process.

Exercise 5 (Basic Model)**.** We model the three possible moves as three conflicting CHR rules:

```
play <=> me(M), select_opp(O), opp(O).
```

where `M` is one of `rock`, `paper` or `scissors`. The constraint `me` represents the move of the player and the constraint `opp` the move of the opponent. `select_opp/1` is a Prolog predicate which chooses one of the three possible moves for the opponent according to a certain strategy. Note that in this model, the strategy of the opponent does not depend on the rule priority (but is decided from outside.

Write the basic program and three `select_opp` predicates:

(1) Only rock is chosen by the opponent.
(2) The opponent chooses randomly between rock and paper.
(3) The opponent chooses randomly between all three possible moves.

Exercise 6 (Recognizing Successes and Failures)**.** The intelligent agent is able to detect a success and a failure. Write CHR rules which recognize win and defeat situations (e.g. if the opponent chose rock and we chose scissors, we lost the round). Count the successes and failures of a move `M` in correspondent `success` and `failure` constraints.

In computational cognitive modeling, production rules often have an associated *utility value* which expresses how successful a rule was in the past. This utility value can be compared with the priorities we introduced in assignment #8. However, the priorities we have used before were only dependent on local values or were completely static.

We translate the concept of utility values to CHR-P: The current utility value of a production rule is assumed to be stored in a constraint `u(R,P)`, where `R` is an identifier of the rule (in our case the move) and `P` is the utility value (i.e. priority). We extend our basic rules from above as follows:

```
u(rock,P) \ play <=> me(rock), select_opp(O), opp(O) : P.
```

Additionally, since the rule with the highest utility value fires, we sort the utilities in the `choose` rule from assignment #8 in descending order.

Exercise 7 (Utility Adaptation)**.** The utility value of a rule depends on its success. A popular formula to adapt the utility of a rule r is the following:

$$U = 20 \cdot P - C$$

where P represents the success probability of a rule (i.e. $P = \frac{\#successes}{\#successes + \#failures}$) and C the costs of a rule. We assume the costs of a rule application to be 1. Extend your program such that it updates the utilities when a success or failure is recognized.

Exercise 8 (Let's play!)**.** Write a Prolog predicate `start` which initializes all utilities, successes and failures to `0` and starts 10 rounds of the game (by adding 10 `play` constraints). Test your model for the different types of opponents.

Rule-based Programming

Prof. Dr. Thom Frühwirth, Daniel Gall

Summer Term 2014

Assignment #10

Ensuring Set-based Semantics

Exercise 1 (Rule Variants)**.** Given any CHR simplification rule (ignoring priorities):

```
H, H1, H2 <=> G | B[,H1,H2].
```

it is possible to generate new rule variants by systematically unifying head constraints in all possible ways:

```
H, H1 <=> H1 = H2, G | B[,H1].
```

The same unify and merge technique can be applied to simpagation and propagation rules; if any of the heads merged was originally to be kept, then it remains kept in the variant rule even if it was unified with a head that would be removed.

Write a rule that transforms a CHR rule written as `chrl/4` constraints into all possible variants, by trying to unify and merge all head constraints. *Hint: Use disjunction in the CHR rule body to enforce possible rule firings to obtain the various unify possibilities.* You might find it useful to encapsulate rule head constraints into `head/2` constraints, where the first argument is the actual head and the second is 1 for kept and 0 for removed constraints. A typical test query would include a rule: `chrl([p(a,X), p(Y,b)],[p(Z,W)],[],[p(X,Z)]).` Test your code with similar such appropriate queries, and show the multiple results.

Logical Algorithms (LA)

An LA rule of the form

$$r \ @ \ [p :] A \rightarrow C$$

can be translated to a CHR rule (with priorities)

$$r \ @ \ A_1 \Rightarrow A_2 \ | \ C \ [: p].$$

where A_1: atoms in A, A_2: built-ins in A. In LA, a set-based semantics is assumed. Thus, additional CHR rules to ensure set-based semantics might be needed.
Additionally, for each LA predicate A the following rules are added to the program:

$$A \ \backslash \ A \Leftrightarrow true.$$
$$del(A) \ \backslash \ del(A) \Leftrightarrow true.$$
$$del(A) \ \backslash \ A \Leftrightarrow true.$$

Exercise 2 (Minimum)**.** The following LA program calculates the minimum:

$$min(X), min(Y), X < Y \rightarrow del(min(Y))$$

Translate the program to CHR. Generate the additional rules to ensure set-based semantics. Are the additional rules needed? Test your program for reasonable examples. What happens, if the predicate $X < Y$ is replaced by $X \leq Y$?

Exercise 3 (Primes)**.** Write an LA program for the prime sieve (similar to the CHR program presented in the lecture). Translate the LA program to CHR and compare the resulting program to the program presented in the lecture. Test your program for queries like:

```
prime(2),prime(3),prime(4),prime(5),prime(6),prime(7).
```

Next exercises are to be submitted by e-mail to: daniel.gall@uni-ulm.de by 09.07.14 at 10:00.

Exercise 4 (Examples for the set-based semantics)**.** For the following list of CHR programs, encode the rule into an equivalent `chrl/4` constraint, then produce all possible rule variants for the set-based semantics. Examine the produced rules and justify if they are necessary, redundant, non-terminating or incorrect.

(1)
```
minimum @ min(Y) \ min(X)<=> Y=<X | true.
```

(2)
```
gcd @ gcd(N) \ gcd(M) <=> (0<N,N=<M) | X is M-N, gcd(X).
```

(3)
```
xor1 @ xor(X), xor(X) <=> xor(0).
xor2 @ xor(0), xor(1) <=> true.
```

(4)
```
primes @ prime(A) \ prime(B) <=> B mod A=:=0 | true.
```

(5)
```
transitive_closure @ p(A,B), p(B,C) ==> true | p(A,C).
```

(6)
```
summing @ accu(X), accu(Y) <=> Z is X+Y, accu(Z).
```

(7)
```
sort @ X <<< A \ X <<< B       <=> A<B | A <<< B.
merge @ merge(N,A), merge(N,B) <=> A<B | M is N+1, merge(M,A), A <<< B.
```

(8)
```
antisymmetry @ leq(X,Y), leq(Y,X)  <=> X = Y.
idempotence  @ leq(X,Y) \ leq(X,Y) <=> true.
transitivity @ leq(X,Y), leq(Y,Z)  ==> leq(X,Z).
```

(9)
```
le @ X le Y, X::A:_, Y::_:D ==> Y::A:D, X::A:D.
eq @ X eq Y, X::A:B, Y::C:D ==> Y::A:B, X::C:D.
ne @ X ne Y, X::A:A, Y::A:A <=> fail.
```

(10)
```
mult_z @ mult(X,Y,Z), X::A:B, Y::C:D ==>
            M1 is A*C, M2 is A*D, M3 is B*C, M4 is B*D,
            Z::min(min(M1,M2),min(M3,M4)):max(max(M1,M2),max(M3,M4)).
```

Exercise 5 (LA Examples)**.** Write an LA program for the gcd and the summing rules from exercise 4 (examples 2 and 6). Translate the program back to CHR.