

### *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). You can use the template file sent to you by e-mail.

- (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 some 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.

---

Knowledge can be processed as “Property of Object is Value”. Examples include:  
`name of X is julia, birth-date of julia is (2,3,1988), value of city is Ulm, ...`  
Add the following two lines to have this knowledge representation:

```
:- op(900, xfx, [of]).  
:- chr_constraint of/2.
```

---

**Exercise 6** (Age Calculator by Propagation). Constraints can be used like a database to store and process information about famous actors. The year of birth of a person can be expressed using like `birth-year of tom-cruise is 1962`. The current year can be expressed as `year of today is 2013`. Write a CHR rule that preserves knowledge already stored, and calculates the current age of an actor. (Assume that the exact day and month do not matter). Your program is correct if it reveals that Tom Cruise is 51 years old! (i.e. `age of tom-cruise is 51`)

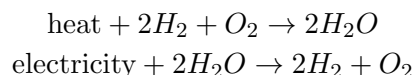
**Exercise 7** (Minimum by Simplification). The age of many people can be represented as `age of tom-cruise is 51, age of arnold-schwarzenegger is 65, ....` Write a single CHR rule that filters these ages to leave only the youngest person.

---

The next exercises are to be submitted by e-mail to: [amira.zaki@uni-ulm.de](mailto:amira.zaki@uni-ulm.de). The deadline is on 06.05.13 by 10:00. 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:



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 as (day,month,year). The current date can be given as `date of today is (25,4,2013)`. Moreover, the date of birth of a person is now represented to also include the day, month and year. 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 50 years old!

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

**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.

**Exercise 2** (Counting Items). You are given as input a sequence of items in `i/1` constraints, for example `i(bread)`, `i(cheese)`, `i(water)`, `i(bread)`. Your task is to count the items. The output is as `n/2` constraints, for example `n(bread,2)`, `n(cheese,1)`, `n(water,1)`.

**Exercise 3** (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:

<code>?- pack(4).</code>	<code>?- pack(25).</code>	<code>?- butt, pack(3).</code>
<code>butt</code>	<code>butt</code>	<code>butt</code>
<code>smoked(5)</code>	<code>smoked(33)</code>	<code>smoked(4)</code>

**Exercise 4** (Geometry). 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 rules that derive further such relationships from given relationships, e.g. `parallel(L1,L2), parallel(L2,L3) ==> parallel(L1,L3)`. Ensure termination.

**Exercise 5** (Hamming's problem - *to be done next week*).

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, ...

Define a non-terminating process `hamming(N)` that will produce the numbers as elements of the infinite chain starting with 1. *Hints*:

- pretend that the sequence is already known
- no base cases in recursion as sequences are infinite
- concurrent-process network, processes can be executed in parallel
- to view the stream, you will need a `watch/2` rule that writes elements of the stream as they are produced

Implement a mixed Prolog and CHR solution then a compact four-ruler pure CHR solution.

---

The next exercises are to be submitted by e-mail to: [amira.zaki@uni-ulm.de](mailto:amira.zaki@uni-ulm.de). The deadline is on 07.05.13 by 10:00. 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 6** (Walking with Big Steps). In the previous assignment, a walk was expressed by four movements `east`, `west`, `south`, `north`. The steps of this walk can be changed to give a number of steps to be walked in a certain direction. Thus a step `west(3)` is equivalent to 3 steps in the west direction, i.e. `west`, `west`, `west`.

Extend the walk program with generic CHR rules that decompose these big steps into single movements. You will need to define new constraints with 1 argument, like `west/1`. For mathematical calculations use `is`, for example `A is B-1`.

Then test it for queries like: `'east(3), south(4), west(3), west(2), south(4), north(2)'`.

**Exercise 7** (CHR-Compactness). Implement the Hamming's problem of exercise#5 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 07.05.13.

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

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

can be represented using a `chr1/4` constraint in the form:

`chr1(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 `chr1` constraint by changing the lists to goals. *Hint*: use `numbervars` to pretty print a term by unifying variables (`_123`, `_456`, ...) to more readable variable names (`A`, `B`, ...). Test your program with (but not limited to) queries such as:

- `chr1([], [a,b], [], [c])` represents: `a,b <=> true | c.`
- `chr1([a,b], [], [], [c])` represents: `a,b ==> true | c.`
- `chr1([a], [b], [], [c])` represents: `a \ b <=> true | c.`
- `chr1([a(X)], [b(Y,Z)], [X<Y], [c(Z)])` represents: `a(A) \ b(B,C) <=> A<B | c(C).`

**Exercise 2** (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 3** (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: [amira.zaki@uni-ulm.de](mailto:amira.zaki@uni-ulm.de) by 21.05.13 by 10:00.

---

**Exercise 4** (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 5** (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)`)

### *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  $X_1, \dots, X_n$  are new variables.:

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

*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:

$$S \rightarrow T$$

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

$$\text{flatten}(X \text{ eq } S) \Leftrightarrow \text{flatten}(X \text{ eq } T)$$

The output CHR rule can be encoded in a `chr1/4` constraint as in assignment#3, thus as:

$$\text{chr1}([], [\text{flatten}(X \text{ eq } S)], [], [\text{flatten}(X \text{ eq } T)])$$

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:

$$\begin{aligned} 0+Y &\rightarrow Y. \\ s(X) + Y &\rightarrow s(X+Y). \end{aligned}$$

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  $\text{and}(X, Y)$  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,  $\text{neg}$ , and disjunction,  $\text{or}$ , 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). Test the correctness of the TRS programs produced by adding an evaluation rule for an  $\text{eq}/2$  constraint, which fires when triggered. A test query must be flattened before fed to the TRS program, hence write a rule for the constraint  $\text{evaluate}(\text{Query})$  which flattens the  $\text{Query}$  then triggers the TRS evaluation.

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  $\text{fd}$  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  $T1 \text{ eq } X$  and  $T2 \text{ eq } Y$ . Thus variants of the existing CHR rules must be created, where head constraints have been unified such that the rules apply after the  $\text{fd}$  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: [amira.zaki@uni-ulm.de](mailto:amira.zaki@uni-ulm.de) by 27.05.13 by 10:00.

---

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

### **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:

$$S \text{ --> } (G) \mid T$$

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

$$X \text{ eq } S \text{ <=> } G \mid \text{flatten}(X \text{ eq } T)$$

where  $X$  is a new variable. The rule should write the output CHR simplification rules to the console, by using translating the FP rule to a `chr1/4` constraint and triggering the writer from assignment#3. 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  $S \text{ --> } T$ .

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 constraints 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). Recall that the first two numbers in the Fibonacci sequence are 0 and 1, and each subsequent number is the sum of the previous two. 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 it for obtaining several of the Fibonacci numbers.

---

Next exercises are to be submitted by e-mail to: [amira.zaki@uni-ulm.de](mailto:amira.zaki@uni-ulm.de) by 03.06.13 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?



## **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:

$$d(x_1), \dots, d(x_n) \iff c(x_1, \dots, x_n) \mid f(x_1, \dots, x_n)$$

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:

$$f(x_1, \dots, x_n) \iff G \mid D, d(y_1), \dots, d(y_m).$$

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.

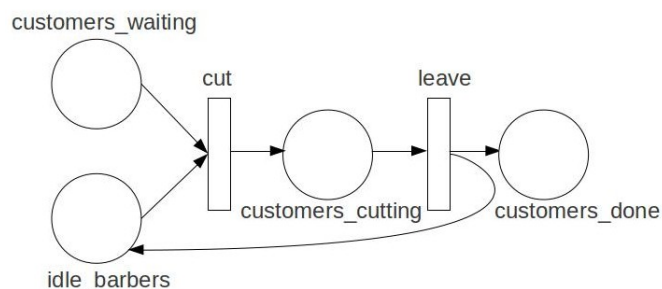
**Exercise 4** (Translating CHR to GAMMA - Walk). Assume we describe a walk (a sequence of steps) by giving directions, `east`, `west`, `south`, `north`. A description of a walk is just a sequence of these CHR constraints. With simplification rules, we can model the fact that certain steps (like `east`, `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.

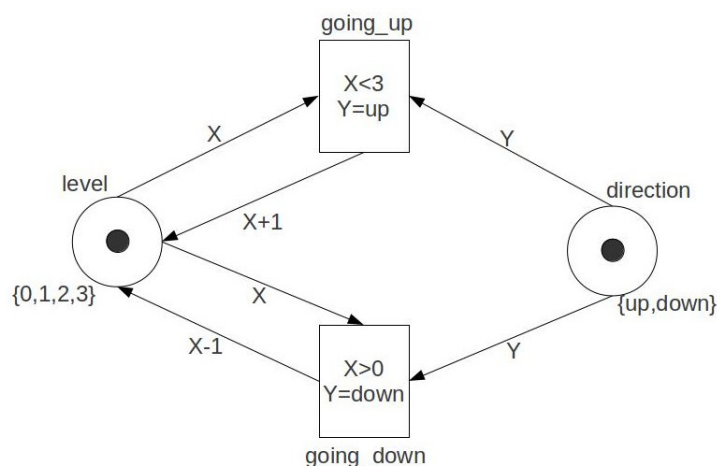
## Petri Nets

**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: [amira.zaki@uni-ulm.de](mailto:amira.zaki@uni-ulm.de) by 12.06.13 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.

### *Production Rule Systems - OPS5*

**Exercise 1** (Preprocessing OPS5). 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>)
  )
```

We define a slight preprocessing step on the input OPS5 rules to make it more readable through Prolog by removing reserved Prolog operators and introducing lists. This involves:

- removing the arrow -->
- changing all rounded and curly brackets to square brackets, i.e. ( ) and { } to [ ]
- changing all spaces to commas ,
- removing attribute markers ^
- removing variable markers < > and instead indexing all variables with a capital V to translate them into valid Prolog variable names
- removing control statements such as (halt) and (crlf)

Applying this preprocessing step on the previous Fibonacci example, it becomes:

```
[p, next_fib,
  [limit, is, Vlimit],
  [[fibonacci, index, [Vi =< Vlimit],
    this-value, Vv1,
    last-value, Vv2], Vfib],
 [modify, Vfib, index, [compute, Vi + 1],
  this-value, [compute, Vv1 + Vv2],
  last-value, Vv1],
 [write, fib, Vi, is, Vv1]
]
```

Apply this preprocessing technique on the **Greatest Common Divisor** example of the lecture.

**Exercise 2** (Translating OPS5 Production Rules to CHR). An OPS5 production rule:

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

which after applying the preprocessing step becomes:

```
[p,N,LHS,-->,RHS]
```

translates to the CHR generalized simpagation rule:

$$N @ LHS1 \setminus LHS2 \leq => LHS3 \mid RHS'$$

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, and RHS': RHS without removal (for LHS2 facts).

Write a rule for a **translate/2** constraint that takes a list containing the preprocessed OPS5 rule and transforms it to an equivalent CHR program.

### *CHR Rules with Negation as Absence*

**Exercise 3** (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:

$N @ Hk \setminus Hr \Leftrightarrow G \mid B : (\text{Neg}B, \text{Neg}C).$

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

$N1 @ Hk, Hr \Leftrightarrow G \mid \text{check}(Hk, Hr).$

$N2 @ \text{Neg}C \setminus \text{check}(Hk, Hr) \Leftrightarrow \text{Neg}B \mid \text{true}.$

$N3 @ Hk \setminus Hr, \text{check}(Hk, Hr) \Leftrightarrow G \mid B.$

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

$\text{num}(X) \Rightarrow \text{min}(X) : (Y < X, \text{num}(Y)).$

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 5** (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:

$N @ Hk \Rightarrow G \mid Bc : (\text{Neg}B, \text{Neg}C).$

This translates to the following pair of CHR rules:

$Nn @ \text{Neg}C \setminus Bc \Leftrightarrow \text{Neg}B \mid \text{true}.$

$Np @ Hk \Rightarrow G \mid B.$

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 6** (Special Negated Examples). Rewrite the three programs of exercise 4 into the special case form and produce their transformed CHR programs. Test the resultant programs with appropriate examples.

### *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:

$\text{Head1} \setminus \text{Head2} \text{ <=> Guard} \mid \text{Body} : P.$

Hence the extended rule can be represented using a `chr1/5` constraint, where `Head1`, `Head2`, `Guard`, `Body` are lists:

`chr1(Head1, Head2, 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,Head1,Head2,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*). Enhance your program such that the additional conflict resolution rules (`collect`, `choose`) are added to an output file. Consider the case when the conflict set does not contain any more rules, how would the previous two rules be modified? Then consider the case when the `choose/3` cannot find any applicable rule from the list, how would this change the required additional rules?

**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)  $P = \text{random}$ : randomly selects a rule from the conflict set
- (2)  $P = \text{bfs}$ : selects a rule from the set to ensure the breadth first traversal of rules
- (3)  $P = N, \text{number}(N)$ : selects the rule with the highest priority ( $P$ )
- (4)  $\text{neg}(C, G)$ : 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:

$\text{d2} @ \text{dist}(X,N) \setminus \text{dist}(X,M) \text{ <=> } N < M \mid \text{true} : 1.$   
 $\text{dn} @ \text{dist}(X,N), \text{edge}(X,Y,M) \text{ ==> } P \text{ is } N+2 \mid Z \text{ is } N+M, \text{dist}(Y,Z) : P.$

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 `chr1/5` 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: [amira.zaki@uni-ulm.de](mailto:amira.zaki@uni-ulm.de) by 25.06.13 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 `chr1/5` 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 `chr1/5` 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.

### *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.



## Ensuring Set-based Semantics

**Exercise 5 (Rule Variants).** Given any CHR simplification rule (ignoring priorities):

$H, H1, H2 \Leftarrow G \mid B[,H1,H2]$ .

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

$H, H1 \Leftarrow H1 = H2, G \mid B[,H1]$ .

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 `chr1/4` 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: `chr1([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.

---

Next exercise is to be submitted by e-mail to: [amira.zaki@uni-ulm.de](mailto:amira.zaki@uni-ulm.de) by 02.07.13 at 10:00.

---

**Exercise 6 (Examples).** For the following list of CHR programs, encode the rule into an equivalent `chr1/4` constraint, then produce all possible rule variants. 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 1.** Constraint Logic Programming (CLP)

Pure Prolog and CLP (without cut and negation-as-failure) can be translated to CHR. A CLP predicate  $p/n$  is considered as CHR constraint, and for each predicate  $p/n$  Clark's completion of  $p/n$  is added as  $\text{CHR}^\vee$  simplification rule. Write a rule for a `translate/2` constraint that transforms a CLP program into CHR. Then translate the following two examples into CHR:

- (1) Append:
- ```
append([],L,L).
append([H|L1],L2,[H|L3]):- append(L1,L2,L3).
```
- (2) Member:
- ```
mem(X,[X|T]).
mem(X,[H|T]):- mem(X,T).
```

**Exercise 2.** Concurrent Constraint Programming (CC)

To embed in CHR, the following transformations take place:

- CC predicates  $\rightarrow$  CHR constraints
- CC constraints  $\rightarrow$  CHR built-in constraints
- CC declaration  $\rightarrow$  CHR simplification rule
- CC agent  $\rightarrow$  CHR goal
- CC ask expression  $\rightarrow$  CHR simplification rules for auxiliary unary CHR constraint ask
- Ask constraint  $\rightarrow$  built-in in guard of CHR rule
- Tell constraint  $\rightarrow$  built-in in body of CHR rule

Write a rule for a `translate/2` constraint that transforms a CC program into CHR. Then translate the following two examples into CHR:

- (1) Maximum:
- ```
max(X,Y,Z) <- (X=<Y -> Y=Z) + (Y=<X -> X=Z)
```
- (2) The Hamming sequence:
- ```
hamming(S) <- S1=[1|S],
               mults(S1,2,S2), mults(S1,3,S3), mults(S1,5,S5)),
               merge(S2,S3,S23), merge(S5,S23,S)

mults([X|Xs],N,L) <- L = [X*N|XsN], mults(Xs,N,XsN)

merge([X|Xs], [Y|Ys], XYOut) <-
  (X=Y -> (XYOut = [X|Out], merge(Xs,Ys,Out)))
+ (X<Y -> (XYOut = [X|Out], merge(Xs,[Y|Ys],Out)))
+ (X>Y -> (XYOut = [Y|Out], merge([X|Xs],Ys,Out)))
```