Part I
CHR tutorial

We present the essentials of the Constraint Handling Rules (CHR) programming language by the use of examples in this Tutorial part.

The first chapter **Getting started** is a step-by-step introduction to CHR based on simple examples.

The second chapter **My first CHR programs** introduces some simple, but concise and effective, CHR programs. We discuss basic properties of CHR programs in an informal way: anytime and online algorithm property, correctness, confluence, concurrency, and complexity. The formal analysis of these programs is deferred to Part III.

Exercises and selected solutions are given for the practical programming chapters in Parts I and III. More exercises can be found online.

1
Getting started

This chapter is a basic introduction to CHR using simple examples. They introduce the types of rules used in CHR, their behavior and basic ingredients of the language such as logical variables and built-in constraints. Last but not least, we define the concrete syntax of CHR and we informally describe how CHR executes its rules.

In this book, we will use the *concrete syntax* of CHR with Prolog as the host language in the practical programming parts and mathematical *abstract syntax* in the formal Part II.

1.1 How CHR works

For programming, we recommend using a CHR implementation from K.U. Leuven, since they are currently the most recent and advanced. The CHR rules themselves will also be executable in other Prolog implementations of CHR and with minor modifications in K.U. Leuven JCHR, an implementation of CHR in Java and in the K.U. Leuven CHR library for C.

When we write a CHR program, we can mix host language statements and CHR code. The CHR-specific part of the program consists of declarations and rules.

1.1.1 Propositional rules

We start programming in CHR with rules that only involve *propositions*, i.e. constraints without arguments. Syntactically, constraints are similar to procedure calls.

Example 1.1.1 (Weather) Everybody talks about the weather, and we do as well.

Declarations. They introduce the CHR constraints we are going to define by the rules. They are specific to the implementation and the host language. Later in this book, we will usually skip the declarations and concentrate on the rules.

We introduce three CHR constraints named `rain`, `wet`, and `umbrella` with the following declaration:

```
:- module(weather, [rain/0]).
:- use_module(library(chr)).

:- chr_constraint rain/0, wet/0, umbrella/0.
```

In the first line, the optional Prolog module declaration puts the CHR program into a module named `weather`, which exports only the mentioned constraint `rain/0`. The *functor* notation `c/n` defines the name (`c`) and number of arguments (`n`) of a constraint $\mathtt{c(t_1,\ldots,t_n)}$. In the host language Prolog and its CHR libraries, function symbols (including constants) start with lower-case letters.

The second line of the declaration makes sure that the CHR library is loaded before any CHR-specific code. CHR constraints must be declared using the `chr_constraint` keyword before they are defined and used by rules. In the declaration, for each CHR constraint, at least its name and arity (number of arguments taken) are given, optional specifications are the input/output mode and the type of the arguments.

Rules. Any kind of constraint-handling rule has an optional name, a left-hand side (l.h.s.) called the *head* together with an optional *guard*, and a right-hand side (r.h.s.) called the *body*. There are three kinds of rules. The head, guard, and body of a rule consist of constraints.

The following two CHR rules encode statements about rain:

```
rain ==> wet.
rain ==> umbrella.
```

The first rule says “If it rains, then it is wet”. The second rule can be read as “If it rains, then we need an umbrella”. Both rules have a head consisting of the constraint `rain`. The bodies are `wet` and `umbrella`, respectively. There are no guards. These rules are so-called *propagation rules*, recognizable by `==>`. These kind of rules do not remove any constraints, they just add new ones.

Queries. Computation of a CHR program is initiated by posing a *query*. The rules of the program will be applied to the query until exhaustion, i.e. until no more change happens. The rule applications will manipulate the query by removing CHR constraints and by adding constraints. The result of the computation is called the *answer*, it simply consists of the remaining constraints.

If we pose the query `rain`, the answer will be `rain, wet, umbrella` (not necessarily in that order).

In Prolog CHR implementations, the query is typically entered at the command line prompt followed by a dot. After the computation has finished, the answer is displayed.

Top-down execution. If we write instead the two simplification rules

```
rain <=> wet.
rain <=> umbrella.
```

then the answer to `rain` will be just `wet`. The first rule is applied and removes `rain`.

Rules are tried in textual order, in a top-down fashion, and so only the first rule will ever be applied in our example. In general, whenever more than one rule application is possible, one rule application is chosen. A rule application cannot be undone (unlike Prolog). We thus say that CHR is a *committed-choice language*.

With propagation rules, we can draw conclusions from existing information. With simplification rules, we can simplify things, as we will see. Simplification rules can also express *state change*, i.e. *dynamic behavior* through updates.

Example 1.1.2 (Walk) Assume we describe a walk (a sequence of steps) by giving directions, `left, right, forward, backward`. A description of a walk is just a sequence of these CHR constraints, and of course, multiplicities matter. Note that the order in which the steps are made is not important to determine the position reached. With simplification rules, we can model the fact that certain steps (like `left` and `right`) 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.

```
left, right <=> true.
forward, backward <=> true.
```

So the walk `left, forward, right, right, forward, forward, backward, left, left` posed as a query yields as answer the simplified and shorter walk `left, forward, forward`.

1.1.2 Logical variables

Declarative programming languages such as CHR feature a special kind of variables. These so-called *logical variables* are similar to mathematical unknowns and variables in logic. A logical variable can be either *unbound* or *bound*. A bound variable is indistinguishable from the value it is bound to. A bound logical variable cannot be overwritten with another value (but it can be bound to the same value again). We call languages with such variables *single-assignment* languages, while more traditional languages like Java and C feature *destructive (multiple) assignment* where the value of a variable can be overwritten.

In Prolog CHR implementations, variable names start with an upper-case letter. The underscore symbol denotes an unnamed variable.

Example 1.1.3 (Men and women) Computer science textbooks always have an example involving people of the two sexes. In our case, we have several men, e.g. `male(joe),...`, and several women `female(sue),...` at a dancing lesson. So we have the CHR constraints `male` and `female`. The two constraints have one argument, the name of the person of that sex. We want to assign men and women for dancing. This can be accomplished by a simplification rule with two head constraints:

```
male(X), female(Y) <=> pair(X,Y).
```

The variables of the rule are `X` and `Y`, they are placeholders for the actual values from the constraints that match the rule head. The scope of a variable is the rule it occurs in. Given a query with several men and women, the rule will pair them until only persons of one sex remain. Clearly the number of pairs is less than the number of men and women.

Types of rules. If we replace the simplification rule by a propagation rule, we can compute all possible pairings, since the `male` and `female` constraints are kept.

```
male(X),  female(Y) ==> pair(X,Y).
```

Now the number of pairs is quadratic in the number of people. Propagation rules can be expensive, because no constraints are removed and thus every combination of constraints that match the rule head may lead to a rule application.

There may also be a dance where a single man dances with several women, and this can be expressed by a *simpagation rule*:

```
male(X) \ female(Y) <=> pair(X,Y).
```

In this type of CHR rule, the constraints left of the backslash `\` are *kept* but the remaining constraints of the head, right of the backslash, are *removed*.

It is worth experimenting with queries where men and women are in different orders to understand how CHR rules execute.

Example 1.1.4 (Family relationships I) The following propagation rule named `mm` expresses that the mother of a mother is a grandmother. The constraint `grandmother(joe,sue)` reads as “The grandmother of Joe is Sue”. The use of variables in the rule `mm` should be obvious.

```
mm @ mother(X,Y), mother(Y,Z) ==> grandmother(X,Z).
```

The rule allows us to derive the grandmother relationship from the mother relationship. For example, the query `mother(joe,ann), mother(ann,sue)` will propagate `grandmother(joe,sue)` using rule `mm`.

1.1.3 Built-in constraints

In CHR, we distinguish two kinds of constraints: *CHR constraints*, which are declared in the current program and defined by CHR rules, and *built-in constraints* (short: *built-ins*), which are predefined in the host language or imported CHR constraints from some other module.

On the left-hand side of a rule, CHR and built-in constraints are separated into head and guard, respectively, while on the right-hand side, the body, they can be freely mixed. The reason is that the head and guard constraints are treated differently when the rule is executed, as we will see soon.

Example 1.1.5 (Family relationships II) The mother of a person is unique, she or he has only one mother.

Syntactic equality. In mathematical terms, the mother relation is a function, the first argument determines the second argument. We can write a simpagation rule that enforces this *functional dependency*:

```
dm @ mother(X,Y) \ mother(X,Z) <=> Y=Z.
```

The rule makes sure that each person has only one mother by equating the variables standing for the mothers. We use the *built-in syntactic equality* `=`. The constraint `Y=Z` makes sure that both variables have the same value, even if it is yet unknown. We may actually safely and correctly assume that, in the remainder of the computation, the occurrences of one variable are replaced by (the value of) the other variable in the equation.

For example, the query `mother(joe,ann), mother(joe,ann)` will lead to `mother(joe,ann)` (the built-in constraint `ann=ann` is simplified away, because it is always true).

Failure. The query `mother(joe,ann), mother(joe,sue)` will fail, because then Joe would have two different mothers, and the rule `dm` for mother will lead to the syntactic equality `ann=sue`. This built-in equality cannot be satisfied, it fails. The built-in has acted as a test now. Failure aborts the computation (it leads to the answer `no` in most Prolog systems).

Variables in queries and head matching. In CHR, the current constraints must *match* the rule head that serves as a pattern (unlike Prolog). The query constraints may contain variables and the matching is successful as long as these variables are not bound by the matching.

In the query `mother(A,B), mother(B,C)` we will see in the answer also `grandmother(A,C)` by rule `mm`. On the other hand, no rule is applicable to the query `mother(A,B), mother(C,D)`. We may, however, add a built-in equality constraint to the query, i.e. `mother(A,B), mother(C,D), B=C`. Then we also get `grandmother(A,D)`. If we add `A=D` instead, the mother relations are matched the other way round by the head of the `mm` propagation rule and we get `grandmother(C,B)`.

If we add `A=C`, the rule `dm` will apply and add `B=D`, so the complete answer will be `mother(A,B), mother(C,D), A=C, B=D`.

We now give a CHR programming example that involves simple arithmetic built-ins.

Example 1.1.6 (Mergers and acquisitions) Let us move to the commercial world of companies. A large company will buy any smaller company. We use a CHR constraint `company(Name,Value)`, where `Value` is the market value of the company.

Guards. This rule describes the merge–acquisition cycle that we can observe in the real world:

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

The meaning of the arithmetic comparison `Value1>Value2` in the guard should be obvious. A guard basically acts as a test or precondition on the applicability of a rule. Only built-in constraints are allowed in a guard. These built-ins should be simple tests, i.e. the same constructs that occur in conditions of the host language.

For readability of the rule, we have used an in-lined arithmetic expression, `Value1+Value2`, inside a CHR constraint. This works in a similar notation for the host language Java, for the host language Prolog we would have to use the built-in `is` to evaluate the arithmetic expression:

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

After exhaustive application of these rules to some companies, only a few big companies will remain, because the rule is applicable in one way or another to any two companies with different market value. If one company remains, we have a monopoly, otherwise we have an oligopoly. All remaining companies will have the same value.

1.2 CHR programs and their execution

We formally introduce the concrete syntax of CHR and we informally introduce the operational semantics of the language.

1.2.1 Concrete syntax

The CHR-specific part of a CHR program consists of declarations and rules. Declarations are implementation-specific, for details consult your manual.

Rules. There are three kinds of constraint-handling rules, which can be seen from the following EBNF grammar. Terminal symbols are in single quotes. Expressions in square brackets are optional. The symbol `|` (without quotes) separates alternatives.

```
Rule --> [Name '@']
         (SimplificationRule | PropagationRule | SimpagationRule) '.'

SimplificationRule --> Head           '<=>' [Guard '|'] Body
PropagationRule    --> Head           '==>' [Guard '|'] Body
SimpagationRule    --> Head '\' Head '<=>' [Guard '|'] Body

Head           --> CHRConstraints
Guard          --> BuiltInConstraints
Body           --> Goal

CHRConstraints --> CHRConstraint | CHRConstraint ',' CHRConstraints
BuiltInConstraints -->   BuiltIn | BuiltIn ',' BuiltInConstraints
Goal           --> CHRConstraint | BuiltIn | Goal ',' Goal

Query          --> Goal
```

A `Head` is a comma-separated sequence of `CHRConstraints`. The `Guard` consists of `BuiltInConstraints`. The symbol `'|'` separates the guard (if present) from the body of a rule. (The symbol is inherited from early concurrent languages and should not be confused with the `'|'` used in the EBNF grammar.) The `Body` of a rule is a `Goal`. A `Goal` is a comma-separated sequence of built-in and CHR constraints. In simpagation rules, the backslash symbol `'\'` separates the head of the rule into two parts. A `Query` is simply a goal.

Basic built-in constraints. With Prolog as the host language, we use the following minimal set of predefined predicates as built-in constraints. Some of them we have already discussed in the introductory examples of this chapter. Built-in constraints may be used for auxiliary computations in the body of a rule, such as arithmetic calculations. Built-in constraints that are tests are typically used in the guard of rules. They can either *succeed* or *fail*. We give the name of the built-in and, preceded by `'/'`, the number of its arguments.

- The most basic built-in constraints:
  `true/0` always succeeds.
  `fail/0` never succeeds, i.e. always fails.

- Testing if variables are bound:
  `var/1` tests if its argument is an unbound variable.
  `nonvar/1` tests if its argument is a bound variable.

- Syntactical identity of expressions (infix notation used):
  `=/2` makes its arguments syntactically identical by binding variables if necessary. If this is not possible it fails.
  `==/2` tests if its two arguments are syntactically identical.
  `\==/2` tests if its two arguments are syntactically different.

- Computing and comparing arithmetic expressions (infix notation used):
  `is/2` binds the first argument to the numeric value of the arithmetic expression in the second argument. If this is not possible it fails.
  `</2,=</2,>/2,>=/2,=:=/2,=\=/2` test if both arguments are arithmetic expressions whose values satisfy the comparison.

Because `=/2` and `is/2` bind their first arguments, they should never be used in guards. The built-ins `==/2` and `=:=/2` should be used instead. (However,