驰

Constraint Handling Rules - Getting started

Prof. Dr. Thom Frühwirth | 2009 | University of Ulm, Germany

Table of Contents

Overview

- Basic introduction to CHR using examples
- Rule types and their behavior
- Logical variables and built-in constraints
- Concrete syntax
- Informal description of rule execution in CHR

CHR implementations

- Most recent and advanced implementation: K.U. Leuven (recommended)
- Programs also executable with minor changes in other Prolog implementations of CHR
- K.U. Leuven JCHR: CHR implementation in Java
- K.U. Leuven CHR library for C
- CHR code (declarations and rules) and host language statements mixed in programs

Declarations

Declarations introduce CHR constraints we will define by rules

Example (Declarations)

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

- Functor notation `c/n`: name `c`, number of arguments `n` of constraint `c` $(t_1, \ldots, t_n)$
- First line: optional Prolog module declaration: declares module `weather`, where only constraint `rain/0` is exported.
- Second line: loading CHR library
- Third line: Defines CHR constraints `rain`, `wet`, and `umbrella`
  - At least name and arity must be given

Rules (I)

- Parts of a rule:
  - Optional name
  - Left-hand side (l.h.s.) called head, with optional guard
  - Right-hand side (r.h.s) called body
- Head, guard, and body consist of constraints
- Three different kind of rules

Rules (II)

Example (Rules)

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

- First rule: “If it rains, then it is wet”
- Second rule: “If it rains, we need an umbrella”
- Head of both rules is `rain`
- Bodies: `wet` and `umbrella`
- No guards
- Also called propagation rules (`==>`)
  - Do not remove constraints, only add new ones

Queries

- Posing query initiates computations
- Rules applied to query until exhaustion (no more changes happen)
- Rule applications manipulate query by removing and adding constraints
- Result (called answer) consists of remaining constraints

Example (Query)

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

Posing query `rain` results in `rain`, `wet`, `umbrella` (not necessarily in this order)

Top-down execution

- Rules applied in textual order
- In general: If more than one rule applicable, one rule is chosen
- Rule applications cannot be undone like in Prolog
  $\Rightarrow$ CHR is a committed-choice language

Example (Top-down execution)

Two simplification rules

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

- Application of first rule removes `rain`
- Second rule never applied

Simplification rules

- Propagation rules
  - Drawing conclusions from existing information
- Simplification rules
  - Simplify things
  - Express state change
  - Dynamic behavior

Example

Example (Walk)

- Walk expressed by movements `east`, `west`, `south`, `north`
- Multiplicity of steps matters, order does not matter for walk
- Simplification rules express that steps can cancel out each other (i.e. `east` and `west`)

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

- Rules simplify walk to one with minimal number of steps
- Query `east, south, west, west, south, south, north, east, east`
  yields answer `east, south, south`

Logical variables

Logical variables

- Featured in declarative languages like CHR
- Similar to mathematical unknowns and variables in logic
- Can be unbound or bound
- Bound variables indistinguishable from value they are bound to
- Bound variables cannot be overridden
- Languages with such variables called single-assignment languages
- Other languages like C and Java feature destructive (multiple) assignments

Example

Example

- Two constraints with one argument representing men (e.g. `male(joe)`) and women (e.g. `female(sue)`)
- Assigning men and woman for dancing with simplification rule

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

- Variables `X`, `Y` placeholders for values of constraints matching rule head
- Scope of variable is rule it appears in
- Given query with several men and women, rule pairs them until only people of one sex left

Types of rules

Example (Propagation rule)

- Computing all possible pairs with propagation rule (keeps `male` and `female` constraints)

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

- Number of pairs quadratic in number of people
  $\Rightarrow$ Propagation rule can be expensive

Example (Simpagation rule)

- One man dances with several women expressed by simpagation rule

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

- Head constraints left of backslash \ kept,
  head constraints right of backslash removed

Example

Example (Family relationships (I))

- Propagation rule named `mm` expresses grandmother relationship

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

- Constraint `grandmother(joe,sue)` reads as “Grandmother of Joe is Sue”
- Allows derivation of grandmother relationship from mother relationship
- `mother(joe,ann), mother(ann,sue)` will propagate `grandmother(joe,sue)` using rule `mm`

Built-in constraints

- Two kinds of constraints in CHR
- CHR constraints (user-defined constraints)
  - Declared in current program and defined by CHR rules
- Built-in constraints (built-ins)
  - Predefined in host language or imported CHR constraints from other modules
- On left hand side CHR and built-ins constraints separated into head and guard
- On right hand side freely mixed

Syntactic equality

Example (Family relationships (II))

- Mother of a person is unique, expressed by rule

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

- Syntactic equality: Mother relation is function, first argument determines second
- Rule enforces this using built-in syntactic equality =
  - Constraint `Y=Z` makes sure that both variables have the same value
  - Occurrences of one variable are replaced by (value of) other variable
- Query `mother(joe,ann), mother(joe,ann)` will lead to `mother(joe,ann)`
  - `ann=ann` simplified away, is always *true*

Failure

Example (Family relationship (III))

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

- Query `mother(joe,ann), mother(joe,sue)` fails (Joe would have two different mothers)
- Rule `dm` will lead to `ann=sue`, which cannot be satisfied
- Built-in acts as test in this case

- Failure aborts computation
- Failure leads to answer `no` in most Prolog systems

Variables in queries and head matching

- Query can contain variables (matching successful as long as they are not bound by matching)

Example (Family relationship (IV))

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

- Answer `grandmother(A,C)` for query `mother(A,B), mother(B,C)`
- No rule applicable to `mother(A,B), mother(C,D)`
- Answer `grandmother(A,D)` when built-in added to query: `mother(A,B), mother(C,D), B=C`
- Adding `A=D` instead leads to `grandmother(C,B)`
- Adding `A=C` makes rule `dm` applicable, leads to `mother(A,B), A=C, B=D`

Example (I)

Example (Mergers and acquisitions)

- CHR constraint `company(Name,Value)` represents company with market value `Value`
- Larger company buys company with smaller value expressed by rule

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

- Guard `Value1>Value2` acts as precondition of rule applicability
- Only built-ins allowed in guard

Example (II)

Example (Mergers and acquisitions cont.)

- In line arithmetic expression `Value1+Value2` works for host language Java
- In Prolog `is` has to be used leading to rule

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

- Rule is applicable to any pair of companies with different value
- After exhaustive only a few companies will remain (all with the same value)

Concrete Syntax

- CHR-specific part of program consists of declarations and rules
- Declarations are implementation-specific
- In following EBNF grammar:
  - Terminals in single quotes
  - Expressions ins square brackets optional
  - Alternatives separated by |

Rules

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

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

- Three different types of rules in CHR
- `'|'` separates guard from body of rule
- `'\'` separates head of simpagation rule into two parts

Rules

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

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

Query             --> Goal
```

- Head of rule is sequence of CHR constraints
- Guard is a sequence of built-ins constraints
- Body is a sequence of built-ins and CHR constraints

Basic built-in constraints (I)

- Using set of predicates from host language Prolog
- Can be used for auxiliary computations in rule body
- Built-ins in guard of rule usually test (succeed or fail)
- Most basic built-ins
  - `true/0` always succeeds
  - `fail/0` never succeeds
- Testing if variables are bound
  - `var/1` tests if argument is unbound variable
  - `nonvar/1` tests if argument is bound variable

Basic built-in constraints (II)

- Syntactical identity of expressions (infix):
  - `=/2` makes arguments syntactically identical by binding variables (fails if binding not possible)
  - `==/2` tests if arguments syntactically identical
  - `\==/2` tests if arguments syntactically different
- Computing and comparing arithmetic expressions (infix):
  - `is/2` binds first argument to value of arithmetic expression in the second argument (fails if not possible)
  - `</2,=</2,>/2,>=/2,=:=/2,=\=/2` test if arguments are arithmetic expressions whose values satisfy comparison

Basic built-in constraints (III)

- `=/2` and `is/2` bind first argument
  $\Rightarrow$ should never be used in guards
- Use `==/2` and `=:=/2` instead
- But some compilers make silent replacement

Informal semantics

- Description of current sequential implementation
- Based on so-called refined operational semantics
- Maybe different rule application in parallel, experimental and future implementations
- Those implementations will still respect so-called abstract operational semantics

Constraints

- Constraint is active operation as well as passive data
- Constraints in goals processed from left to right
- When CHR constraint encountered:
  - Evaluated like procedure call
  - Checks applicability of rules it appears in
  - Called active constraint
- Rules applied in textual order
- If no rule applicable to active constraint it becomes passive and is put in constraint store
- Passive constraints become active again context changes (their variables get bound)

Head matching

- One head constraint of rule is matched against active constraint
- Matching succeeds if constraint serves pattern
- Matching may bind variables in head (not in active constraint)
- If matching succeeds and rule head consists of more than one constraint, constraint store is searched for partner constraints to match other head constraints

- Head constraints searched from left to right
- Exception: simpagation rule
  - Constraints to be removed searched for before constraints to be kept are searched for
- If matching succeeds, guard is checked
- If several head constraints match active constraint, rule tried for each matching
- If no successful matching exists, active constraint tries next rule

Guard checking

- Guard is precondition on rule applicability
- Test that either succeeds or fails
- If guard succeeds, rule is applied
- If guard fails, active constraint tries next head matching

Body execution

- When rule is applied, we say it fires
- Simplification rule: matching constraints removed, body executed
- Simpagation rule: similar to simplification rule but constraints matching head part preceding $\backslash$ kept.
- Propagation rule: Body executed without removing any constraints
- Propagation rule will not fire with same constraint again
- According to rule type head constraints either called kept or removed
- Next rule tried when active constraint not removed