CHR - a common platform for rule-based approaches

馳

Prof. Dr. Thom Frühwirth | June 2010 | Uni Ulm

Renaissance of rule-based approaches

Results on rule-based system re-used and re-examined for

- Business rules and Workflow systems
- Semantic Web (e.g. validating forms, ontology reasoning, OWL)
- UML (e.g. OCL invariants) and extensions (e.g. ATL)
- Computational Biology
- Medical Diagnosis
- Software Verification and Security

Overview

Embedding rule-based **approaches** in CHR

Using source-to-source transformation (no interpreter, no compiler)

- Rewriting- and graph-based **formalisms**
  - Term Rewriting Systems
  - Chemical Abstract Machine and Multiset Transformation
  - Colored Petri Nets
- Rule-based **systems**
  - Production Rules
  - Event-Condition-Action Rules
  - Logical Algorithms
- Logic- and constraint-based **programming languages**
  - (Deductive Databases)
  - Prolog and Constraint Logic Programming
  - Concurrent Constraint Programming

Embeddings in CHR
Advantages

- Advantages of CHR for **execution**
  - Efficiency, also optimal complexity possible
  - Abstract execution by constraints, even when arguments unknown
  - Incremental, anytime, online algorithms for free
  - Concurrent, parallel for confluent programs
- Advantages of CHR for **analysis**
  - Decidable confluence and operational equivalence
  - Estimating complexity semi-automatically
  - Logic-based declarative semantics for correctness
- *Embedding allows for comparison and cross-fertilization (transfer of ideas)*

Potential shortcomings of embeddings in CHR

⇒ Use extensions of CHR (dynamic CHR covers all

- for built-in “**negation**” of rb systems, deductive db and Prolog
  ⇒ CHR with negation-as-absence
- for **conflict resolution** of rule-based systems
  ⇒ CHR with priorities
- for built-in **search** of Prolog, constraint logic programming
  ⇒ CHR with disjunction or search library
- for ignorance of **duplicates** of rule-based formalisms
  ⇒ CHR with set-based semantics
- for **diagrammatic notation** of graph-based systems
  ⇒ CHR with graphical interface

Instead of extensions, special-purpose CHR programs can be used.

Positive ground range-restricted CHR

- All approaches can be embedded into simple CHR fragment (except Prolog, constraint logic programming)
  - **ground**: queries ground
  - **positive**: no built-ins in body of rule
  - **range-restricted**: variables in guard and body also in head
- These conditions imply
  - Every state in a computation is ground
  - CHR constraints do not delay and wake up
  - Guard entailment check is just test
  - Computations cannot fail
- Conditions can be relaxed: auxiliary functions as non-failing built-ins in body

Distinguishing features of CHR for programming

Unique combination of features

- **Multiple Head Atoms** not in other programming languages
- **Propagation rules** only in deductive db, Logical Algorithms
- **Constraints** only in constraint-based programming
  - Logical variables instead of ground representation
  - Constraints are reconsidered when new information arrives
  - Notion of failure due to built-in constraints
- **Logical Declarative Semantics** only in logic-based prog.
  - CHR computations justified by logic reading of program

Embedding fragments of CHR in other rule-based approaches

Possibilities are rather limited (without interpreter or compiler)

- **Positive ground range-restricted fragment** embeddable into
  - Rule-based systems with negation and Logical Algorithms
  - Only simplification rules in Rewriting- and Graph-based approaches (except Petri-nets)
  - Only propagation rules in deductive databases
- **Single-headed rules** embeddable into
  - Concurrent constraint programming languages

Rewriting-based and graph-based formalisms

Embedding of classical computational formalisms in CHR

- States mapped to CHR constraints
- Transitions mapped to CHR rules

Results in certain types of **positive ground range-restricted CHR simplification rules (PGRS rules)**

Rewriting-based and graph-based formalisms (I)

- Term rewriting systems (TRS)
  - Replace subterms given term according to rules until exhaustion
  - Analysis of TRS has inspired related results for CHR (termination, confluence)
  - Formally based on equational logic
- Functional Programming (FP)
  - Related to syntactic fragment of TRS extended with built-ins
- Graph transformation systems (GTS)
  - Generalise TRS: graphs are rewritten under matching morphism

Rewriting-based and graph-based formalisms (II)

- GAMMA
  - Based solely on multiset rewriting
  - Basis of Chemical Abstract Machine (CHAM)
  - Chemical metaphor of reacting molecules
- Graph-based diagrammatic formalisms
  - Examples: Petri nets, state charts, UML activity diagrams
  - Computation: tokens move along arcs
  - Token at nodes correspond to constraints, arcs to rules

Term rewriting systems (TRS) and CHR

Principles

- Rewriting rules: directed equations between ground terms
- Rule application: Given a term, replace subterms that match lhs. of rule with rhs. of rule
- Rewriting until no further rule application is possible

Comparison to CHR

- TRS locally rewrite subterms at fixed position in one ground term (functional notation)
- CHR globally manipulates several constraints in multisets of constraints (relational notation)
- TRS rules: **no built-ins**, no guards, no logical variables
- TRS rules: restrictions on occurrences of pattern variables

TRS map to subset of positive ground range-restricted simplification rules without built-ins over binary CHR constraint for equality

Flattening

Transformation forms basis for embedding TRS (and FP) in CHR

- Opposite of variable elimination, introduce new variables
- Flattening function transforms atomic equality constraint `eq` between nested terms into conjunction of *flat* equations

Definition (Flattening function)

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

(X variable, T term, $X_1 \ldots X_n$ new variables)

Embedding TRS in CHR

Definition (Rule scheme for term rewriting rule)

TRS rule

$$S \rightarrow T$$

translates to CHR simplification rule

$$[X \ \mathrm{eq} \ S] \Leftrightarrow [X \ \mathrm{eq} \ T]$$

(X new variable, eq CHR constraint)

Example (Addition of natural numbers)

Example (TRS)

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

Example (CHR)

```
T eq T1+T2, T1 eq 0, T2 eq Y <=> T eq Y.
T eq T1+T2, T1 eq s(T3), T3 eq X, T2 eq Y <=>
        T eq s(T4), T4 eq T5+T6, T5 eq X, T6 eq Y.
```

Example (Logical conjunction)

Example (TRS)

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

Example (CHR)

```
T eq and(T1,T2), T1 eq 0, T2 eq Y <=> T eq 0.
T eq and(T1,T2), T1 eq X, T2 eq 0 <=> T eq 0.
T eq and(T1,T2), T1 eq 1, T2 eq Y <=> T eq Y.
T eq and(T1,T2), T1 eq X, T2 eq 1 <=> T eq X.
T eq and(T1,T2), T1 eq X, T2 eq X <=> T eq X.
```

Completeness and nonlinearity

- TRS **linear** if variables occur at most once on lhs. and rhs.
- Translation by flattening incomplete if TRS nonlinear

Example

In the CHR translation, TRS rule `and(X,X) -> X` applicable to `and(0,0)` but not directly to `and(and(0,1), and(0,1))`.

Structure sharing (I)

- *Structure sharing* makes nonlinear but *confluent* TRS complete
- **Confluence:** Given term, each possible rule application sequence leads to same result

Implemented by simpagation rule enforcing functional dependency of `eq` (added at beginning of program)

Definition (Rule for Structure Sharing)

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

Example

```
Z eq and(X,Y), W eq and(X,Y)
```

now reduces to

```
Z eq and(X,Y), W=Z
```

Structure sharing (II)

- Rule `fd` removes equations ⇒ other rules may no longer apply
- Solution: Additional CHR rules, so that rules also apply after application of `fd` (regain confluence)
- Corresponds to enforcing set-based semantics as in LA
  - Transformation applies to CHR rules in general
  - Generation of new rule variants by unifying head constraints

Example

- TRS rule `and(X,X) -> X` translates to
  `T eq and(T1,T2), T1 eq X, T2 eq X <=> T eq X`
- Expects `T1 eq X` and `T2 eq X` even if `T1=T2`; unify them:
- additional rule `T eq and(T1,T1), T1 eq X <=> T eq X`

Functional programming (FP)

- FP can be seen as programming language based on TRS formalism
  - Extended by built-in functions and guard tests
  - Syntactic restrictions on lhs. of rewrite rule: Matching only at outermost redex of lhs

Translation

Definition (Rule scheme for functional program rule)

FP rewrite rule

$$S \rightarrow G \mid T$$

translates to CHR simplification rule

$$X \ \mathtt{eq} \ S \Leftrightarrow G \mid [X \ \mathtt{eq} \ T]$$

(X new variable)

Additional generic rules for data and auxiliary functions

```
X eq T ⇔ datum(T) | X=T.
X eq T ⇔ builtin(T) | call(T,X).
```

(`call(T,X)` calls built-in function `T`, returns result in `X`)

Generic rules can be applied at compile time to body (and head)

Examples (Adding natural numbers, logical conjunction)

Example (Addition of natural numbers in CHR)

```
T eq 0+Y <=> T eq Y.
T eq s(X)+Y <=> T=s(T4), T4 eq T5+T6, T5 eq X, T6 eq Y.
```

Example (Logical conjunction in CHR)

```
T eq and(0,Y) <=> T=0.
T eq and(X,0) <=> T=0.
T eq and(1,Y) <=> T eq Y.
T eq and(X,1) <=> T eq X.
T eq and(X,Y) <=> T eq X.
```

Example (Fibonacci Numbers)

Example (Fibonacci in FP)

```
fib(0) -> 1.
fib(1) -> 1.
fib(N) -> N>=2 | fib(N-1)+fib(N-2).
```

Example (Fibonacci in CHR)

```
T eq fib(0) <=> T=1.
T eq fib(1) <=> T=1.
T eq fib(N) <=> N>=2 | call(F1+F2,T),
                       F1 eq fib(N1), call(N-1,N1),
                       F2 eq fib(N2), call(N-2,N2).
```

(Generic rules for datum and built-in already applied in bodies)

Graph transformation systems (*)

- Can be seen as nontrivial generalization of TRS
  - Instead of terms, graphs are rewritten under matching morphism
- Encoding of GTS production rules exists for CHR (complete, sound)
- Confluence: GTS joinability of critical pairs mapped to joinability of specific critical pairs in CHR

GAMMA

- Chemical metaphor: molecules in solution react according to reaction rules
- Reaction in parallel on disjoint sets of molecules
- Molecules modeled as unary CHR constraints, reactions as rules

Definition (GAMMA)

- GAMMA program: pairs $(c/n, f/n)$ (predicate c, function f)
- f applied to molecules for which c holds
- Result $f(x_1, \dots, x_n) = \{y_1, \dots, y_m\}$ replaces $\{x_1, \dots, x_n\}$ in S
- Repeat until exhaustion

GAMMA Translation

Definition (Rule scheme for GAMMA pair)

GAMMA pair $(c/n, f/n)$ translated to simplification rule

$$d(x_1),\ldots,d(x_n) \Leftrightarrow c(x_1,\ldots,x_n) \mid f(x_1,\ldots,x_n),$$

where f is defined by rules of the form

$$f(x_1,\ldots,x_n) \Leftrightarrow G \mid D, d(y_1),\ldots,d(y_m),$$

(d wraps molecules, c built-in, G guard, D auxiliary built-ins)

Can unfold f if defined by one rule, optimize to simpagation rules
(CHR simplification rules can be translated to GAMMA)

GAMMA examples and translation into CHR

Example (Minimum)

```
min=(</2,first/2)      min @ d(X), d(Y) <=> X<Y | first(X,Y).
                             first(X,Y) <=> d(X).
```

Example (Greatest Common Divisor)

```
gcd=(</2,gcdsub/2)     gcd @ d(X), d(Y) <=> X<Y | gcdsub(X,Y).
                             gcdsub(X,Y) <=> d(X), d(Y-X).
```

Example (Prime sieve)

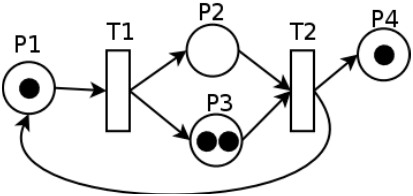
```
prime=(div/2,first/2) prime @ d(X), d(Y) <=> X div Y | first(X,Y).
                                first(X,Y) <=> d(X).
```

Examples can be optimised into single simpagation rules.

Petri nets

- Petri nets consist of
  - Places (P) (○)
  - Tokens (●)
  - Arcs (→)
  - Transitions (T) (||)
- Tokens reside in places, move along arcs through transitions
- Transitions
  - Fire if tokens are present on all incoming arcs:
  - tokens removed from incoming arcs, placed on outgoing arcs

Example (Petri net)



- Places P1 - P4
  - P1 and P4 contain one token, P3 contains two tokens
- Transitions T1 and T2
  - T1 needs one incoming token, produces two outgoing tokens
  - T2 needs two incoming token, produces two outgoing tokens

Colored Petri nets

- Standard Petri nets translate to tiny fragment of CHR
  - Nullary constraints and simplification rules
- **Colored Petri nets**: tokens have different colors
  - Places allow only certain colors
  - Number of colors is fixed and finite
  - Transitions guarded with conditions on token colors
  - Equations at transitions generate new tokens
  - Sound and complete translation to CHR exists

(Colored) Petri Nets are **not** turing-complete.

Colored Petri nets Translation

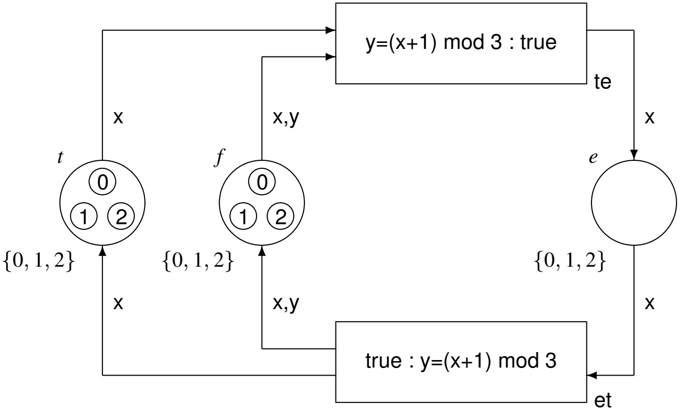
Simplification rules over unary constraints

- Places → unary CHR constraint symbols
- Tokens → arguments of place constraints
- Colors → finite domains (possible values)
- Transitions → CHR simplification rules
  - Incoming arc annotation → rule head
  - Outgoing arc annotation → rule body
  - Transition guard → rule guard
  - Transition equation → rule body

Example – The Dining philosophers Problem

- The dining philosophers problem
  - Philosophers at round table, between each philosopher one fork
  - Philosophers either eat or think
  - For eating, forks from both sides required
  - After eating, philosophers start thinking again
- Dining philosophers as Colored Petri net
  - Philosopher, fork $\rightarrow$ colored tokens
  - Tokens x, y are neighbors at round table if $y = (x + 1) \mod 3$
  - Places: eat `e`, think `t`, fork `f`
  - Arcs: eat-to-think `et` and think-to-eat `te`

Three (3) dining philosophers Colored Petri net



Three dining philosophers CHR translation

Example (Dining philosophers in CHR)

```
te@ t(X),f(X),f(Y) <=> [X,Y]::[0,1,2],Y=(X+1) mod 3 | e(X).
et@ e(X) <=> X::[0,1,2] | Y=(X+1) mod 3, t(X),f(X),f(Y).
```

- `V::L` - variable or variables in list `V` take only values from list `L`
- Query: `t(0),t(1),t(2), f(0),f(1),f(2)`
- Note: `et` rule is reverse of `te` rule (nonterminating)
- Observe loop: add e.g. `e(X) ==> println(e(X))` in front
- Use conflict resolution to obtain fair rule scheduling
- Can be easily generalized to any given finite n
- CHR Rules for Colored Petri Nets are similar to rules for GAMMA (but only finite domains)

Rule-based systems

Overview

Use ground representation

- **Production rule systems**
  - First rule-based systems
  - Imperative, destructive assignment $\Rightarrow$ no declarative semantics
  - Developed in the 1980s
- **Event-Condition-Action (ECA) rules**
  - Extension of production rules
  - For active database systems
  - Hot research topics in the mid-1990s
  - Some aspects standardized in SQL-3

Overview, contd.

- **Business rules**
  - Constrain structure and behavior of business
  - Describe operation of company and interaction with costumers and other companies
  - Recent commercial approach (since end of 1990s)
- **Logical Algorithms formalism**
  - Hypothetical declarative production rule language
  - Similar to Deductive Databases
  - Overshadowing information instead of removal
  - More recent approach (early 2000s)

Production rule systems

Working memory stores facts (working memory elements, WME)

Facts have name and named attributes

Production rule

if *Condition* **then** *Action*

- **If**-clause: *Condition*
  - Expression matchings describing facts
- **Then**-clause: *Action*
  - insertion and removal of facts
  - IO statements
  - auxiliary functions

Production rule systems semantics

Execution cycle

1. Identify all rules with satisfied if-clause
2. **Conflict resolution** chooses one rule
   - e.g. based on priority
3. Then-clause is executed

Continue until exhaustion (all rules applied)

Embedding Production rules in CHR

- **Facts** translate to CHR constraints
  - Attribute name encoded by argument position
- **Production rules** translate to CHR *(generalised) simpagation rules*
  - If-clause forms head and guard, then-clause forms body
- Removal/insertion of facts by positioning in head/body of rule
- **Negation-as-absence** and **conflict resolution** implementable with *refined semantics* or CHR extensions

Translation

Definition (Rule scheme for production rule)

OPS5 production rule

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

translates to CHR generalized simpagation rule

```
N @ LHS1 \ LHS2 ⇔ LHS3 | RHS'
```

`LHS` left hand side (if-clause), `RHS` right hand side (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)

Example (Fibonacci)

Example (OPS5)

```
(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>))
```

Example (CHR)

```
next-fib @ limit(Lim), fibonacci(I,V1,V2) <=> I =< Lim |
    fibonacci(I+1,V1+V2,V1), write(fib I is V1), nl.
```

Example (Greatest common divisor) (I)

Example (OPS5)

```
(p done-no-divisors
    (euclidean-pair ^first <first> ^second 1) -->
    (write GCD is 1) (halt) )

(p found-gcd
    (euclidean-pair ^first <first> ^second <first>) -->
    (write GCD is <first>) (halt) )
```

Example (CHR)

```
done-no-divisors @ euclidean_pair(First, 1) <=> write(GCD is 1).

found-gcd @ euclidean_pair(First, First) <=> write(GCD is First).
```

Example (Greatest common divisor) (II)

Example (OPS5)

```
(p switch-pair
    {(euclidean-pair ^first <first>
                     ^second { <second> > <first>} )
    <e-pair>} -->
    (modify <e-pair> ^first <second>
                     ^second <first>)
    (write <first> -- <second> (crlf)) )
```

Example (CHR)

```
switch-pair @ euclidean_pair(First, Second) <=> Second > First |
    euclidean_pair(Second, First),
    write(First--Second), nl.
```

Example (Greatest common divisor) (III)

Example (OPS5)

```
(p reduce-pair
    {(euclidean-pair ^first <first>
                     ^second { <second> < <first> } )
    <e-pair>} -->
    (modify <e-pair> ^first (compute <first>-<second>))
    (write <first> -- <second> (crlf)) )
```

Example (CHR)

```
reduce-pair @ euclidean_pair(First, Second)) <=> Second < First |
    euclidean_pair(First-Second, Second),
    write(First--Second), nl.
```

Negation-as-absence

Negated pattern in production rules

- Satisfied if no fact satisfies condition
- Violates monotonicity

Example (Minimum in OPS5)

```
(p minimum
    (num ^val <x>)
   -(num ^val < <x>)
   -->  (make min ^val <x>) )
```

Negation-as-absence II

Example (Transitive closure in OPS5)

```
(p init-path
    (edge ^from <x> ^to <y>)
    -(path ^from <x> ^to <y>)
    -->  (make path ^from <x> ^to <y>) )
(p extend-path
    (edge ^from <x> ^to <y>)
    (path ^from <y> ^to <z>)
    -->  (make path ^from <x> ^to <z>) )
```

Default reasoning

- Negation-as-absence can be used for default reasoning
  - Default is assumed unless contrary proven

Example (Marital status in OPS5)

```
(p default
    (person ^name <x>)
    -(married ^name <x>)
    -->
    (make single ^name <x>) )
```

Status `single` is default

Translation of Negation

- Two approaches and one special case
  - Built-in constraints in guard
  - CHR constraint in head
  - Special case: body in head
- Yet another approach: use explicit deletion of ECA rules
- Assume w.l.o.g. one negation per rule (not nested)
- Positive rule parts translated as before

Built-in constraint in guard

Definition (Rule scheme for production rule with negation by built-in)

OPS5 production rule

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

translates to CHR generalized simpagation rule

N @ LHS1 \ LHS2 ⇔ LHS3 ∧ not NEG′ | RHS′

- `LHS`: positive part of lhs
- `LHS1`, `LHS2`, `LHS3`, `RHS′`: as before
- `NEG′`: `NEG` with patterns for facts and conditions wrapped in low-level built-in, `not` negates check
- Built-in checks store for presence of CHR constraint

Examples

Example (Minimum in CHR)

```
minimum @ num(X) ==> not find_c(num(Y),Y<X) | min(X).
```

Example (Transitive closure in CHR)

```
init-path @ e(X,Y) ==> not find_c(p(X,Y),true) | p(X,Y).
```

Example (Marital status in CHR)

```
default @ person(X) ==> not find_c(married(X),true) | single(X).
```

- `find_c(onstraint)`: low-level built-in, makes analysis hard

CHR constraint in head (I)

Definition (Rule scheme for production rule with negation in head)

OPS5 production rule

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

translates to CHR rules

$$\mathtt{N1}\ @\ \mathtt{LHS1} \wedge \mathtt{LHS2} \Rightarrow \mathtt{LHS3} \mid \mathtt{check(LHS1,LHS2)}$$

$$\mathtt{N2}\ @\ \mathtt{NEG1} \setminus \mathtt{check(LHS1,LHS2)} \Leftrightarrow \mathtt{NEG2} \mid \mathit{true}$$

$$\mathtt{N3}\ @\ \mathtt{LHS1} \setminus \mathtt{LHS2} \wedge \mathtt{check(LHS1,LHS2)} \Leftrightarrow \mathtt{RHS'}$$

- `NEG1` patterns, `NEG2` conditions of `NEG`
- `check`: auxiliary CHR constraint
- refined semantics ensures rule `N2` is tried before rule `N3`

CHR constraint in head (II)

Explaination

```
N1 @ LHS1 ∧ LHS2 ⇒ LHS3 | check(LHS1,LHS2)
N2 @ NEG1 \ check(LHS1,LHS2) ⇔ NEG2 | true
N3 @ LHS1 \ LHS2 ∧ check(LHS1,LHS2) ⇔ RHS'
```

- Given `LHS`, check for absence of `NEG` with `check` using `N1`
- If `NEG` found using `N2`, then remove `check`
- **Otherwise** apply rule using `N3` and remove `check`
- Relies on rule order between `N2` and `N3`
- Works under refined semantics or with rule priorities

Examples

Example (Minimum in CHR)

```
num(X) ==> check(num(X)).
num(Y) \ check(num(X)) <=> Y<X | true.
num(X) \ check(num(X)) <=> min(X).
```

Example (Transitive closure in CHR)

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

Example (Marital status in CHR)

```
person(X) ==> check(person(X)).
married(X) \ check(person(X)) <=> true.
person(X)  \ check(person(X)) <=> single(X).
```

CHR rules with negation-as-absence

Definition (Rule scheme for CHR rule with negation in head)

CHR generalised simpagation rule

```
N @ LHS1 \ LHS2 -(NEG1,NEG2) ⇔ LHS3 | RHS
```

translates to CHR rules

```
N1 @ LHS1 ∧ LHS2 ⇒ LHS3 | check(LHS1,LHS2)
N2 @ NEG1 \ check(LHS1,LHS2) ⇔ NEG2 | true
N3 @ LHS1 \ LHS2 ∧ check(LHS1,LHS2) ⇔ RHS
```

- `NEG1` CHR constraints, `NEG2` built-in constraints
- `check`: auxiliary CHR constraint
- refined semantics ensures rule `N2` is tried before rule `N3`
- may not work incrementally when `NEG1` removed later

CHR rules with special-case negation-as-absence

Assume negative part holds, otherwise repair later

- Use `RHS` directly instead of auxiliary `check`
- Works if `RHS` nonempty, no built-ins, contains head variables

Definition (Rule scheme for CHR rule with negation in head)

CHR generalised simpagation rule

`N @ LHS1 \ LHS2 -(NEG1,NEG2)` $\Leftrightarrow$ `LHS3 | RHS`

translates to CHR rules

`N2 @ NEG1 \ RHS` $\Leftrightarrow$ `NEG2 |` *true*

`N3 @ LHS1` $\wedge$ `RHS \ LHS2` $\Leftrightarrow$ *true*

`N1 @ LHS1` $\wedge$ `LHS2` $\Rightarrow$ `LHS3 | RHS`

- If `LHS2` is empty, rule `N3` can be dropped

Special case: body in head

Assume negative part holds, otherwise repair later

- Works if `LHS2` is empty and `RHS` nonempty, contains only CHR constraints and enough `NEG1` variables

Definition (Rule scheme for production rule with special negation)

OPS5 production rule

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

translates to CHR rules

```
Nn @ NEG1 \ RHS′ ⇔ NEG2 | true
Np @ LHS1 ⇒ LHS3 | RHS′
```

Rules are ordered: `Nn` rules have to come before `Np` rules

Consequences and examples

- Shorter, more concise programs, often incremental, concurrent, declarative ⇒ easier analysis
- Negation often not needed (if we have propagation rules)

Example (Minimum in CHR)

```
num(Y) \ min(X) <=> Y<X | true.
num(X) ==> min(X).
```

Example (Transitive closure in CHR)

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

Example (Marital Status in CHR)

```
married(X) \ single(X) <=> true.
person(X) ==> single(X).
```

Simple conflict resolution (I)

Choose rule to be applied among applicable rules.
Assume (total) order for comparing rules.
Implementable for arbitrary CHR rules under refined semantics.

Definition (Rule scheme for CHR rule with static or dynamic weight)

Generalised simpagation rule (with weight, priority or probability `P`)

```
H1 \ H2 ⇔ Guard | Body : P
```

translates to CHR rules

```
H1 ∧ H2 ∧ delay ⇒ Guard | rule(P,H1,H2)
H1 \ H2 ∧ rule(P,H1,H2) ∧ delay ∧ apply ⇔ Body ∧ delay ∧ apply
```

- `delay`: auxiliary constraint to find applicable rules
- `rule`: contains an applicable rule
- `apply`: auxiliary constraint executes chosen rule

Simple conflict resolution (II)

One additional generic rule for rule choice

Rule to resolve conflict

`choose @ rule(P1,_,_) \ rule(P2,_,_)` ⇔ `P1≥P2` | *true*

Phase constraints `delay` ∧ `apply` present (at end of query):

- Constraint `delay` stores applicable rules in `rule`
- Rule `choose` selects rule with largest weight
- Constraint `apply` removes `delay` and executes chosen rule
- **Then** `delay` is called again
- **Then** `apply` is called again

Incremental general conflict resolution (I)

Choose rule to be applied among applicable rules.
Implementable for arbitrary CHR rules under refined semantics.

Definition (Rule scheme for CHR rule with given property)

Generalised simpagation rule (with property `P`)

```
H1 \ H2 ⇔ Guard | Body : P
```

translates to CHR rules

```
delay @ H1 ∧ H2 ⇒ Guard | conflictset([rule(P,H1,H2)])
apply @ H1 \ H2 ∧ apply(rule(P,H1,H2)) ⇔ Body
```

- Rule `delay`: finds applicable rules
- Constraint `conflictset`: collects applicable rules
- Rule `apply`: executes chosen rule

Incremental general conflict resolution (II)

Additional generic rules for rule choice

Rules to resolve conflict

```
collect @ conflictset(L1) ∧ conflictset(L2) ⇔
            append(L1,L2,L3) ∧ conflictset(L3)
choose  @ fire ∧ conflictset(L) ⇔
            choose(L,R,L1) ∧ apply(R) ∧ conflictset(L1) ∧ fire
```

Phase constraint `fire` present (at end of query)

- Rules `delay`, `collect` collect applicable rules in `conflictset`
- Constraint `fire` present: rule `choose` selects rule `R`
  - Rule `R` applied by rule `apply`
  - Updated `conflictset` without applied rule added
  - **Then** `fire` is called again

Summary production rule systems in CHR

- Negation-as-absence and conflict resolution use very similar translation scheme
- Propagation and simpagation rules come handy
- Special case of negation-as-absence avoids negation at all
- Phase constraint avoids rule firing before conflict resolution
- Phase constraints relies on left-to-right evaluation order of queries
- Program sizes are roughly propertional to each other
- CHR complexity roughly as original production rule program

Event Condition Action rules

Extension of production rules for databases, generalise features like integrity constraints, triggers and view maintenance

ECA rules

on *Event* **if** *Condition* **then** *Action*

- *Event*
  - triggers rules
  - external or internal
  - composed with logical operators and sequentially in time
- *(Pre-)condition*
  - includes database queries
  - satisfied if result non-empty
- *Action*
  - include database operations, rollbacks, IO and application calls

Issues in ECA rules

Technical and semantical questions arise

- Different results depending on point of execution.
  Solution: Coupling modes: immediately, later in the same or outside the transaction
- Applied to single tuples or sets of tuples?
- Application order of rules (priorities)
- Concurrent or sequential execution?
- Conflict resolution may be necessary

We choose solution that goes well with CHR

Embedding ECA rules in CHR

- Model events and database tuples as CHR constraints
- Update event constraints `insert/1`, `delete/1`, `update/2`

Definition (Rule scheme for database relation)

n-ary relation r generates CHR rules

```
ins @ insert(R) ⇒ R
del @ delete(P) \ R ⇔ match(P,R) | true
upd @ update(R,R1) \ R ⇔ R1
```

(R= $r(x_1,\ldots,x_n)$, R1= $r(y_1,\ldots,y_n)$, x_i, y_j distinct variables)

`match(P,R)` holds if tuple `R` matches tuple pattern `P`

Additional generic rules to remove events (at end of program)

Definition (Database operation event removal)

`insert(_)` ⇔ *true* `delete(_)` ⇔ *true* `update(_,_)` ⇔ *true*

Example (Salary increase)

Limit employee's salary increase by 10 %

- *Before* update happends (by rule `upd`)

Example

```
update(emp(Name,S1), emp(Name,S2)) <=> S2>S1*(1+0.1) |
                update(emp(Name,S1),emp(Name,S1*1.1)).
```

- *After* update happends (by rule `upd`)

Example

```
update(emp(Name,S1), emp(Name,S2)) <=> S2>S1*(1+0.1) |
                update(emp(Name,S2),emp(Name,S1*1.1)).
```

- Difference: first argument of `update` in the body

More Examples

Production rule examples as ECA rules for database updates

Example (Transitive closure with ECA rules in CHR)

```
insert(p(X,Y)), p(X,Y) ==> delete(p(X,Y)).
insert(e(X,Y)) ==> insert(p(X,Y)).
```

Example (Marital Status with ECA rules in CHR)

```
insert(married(X)), single(X) ==> delete(single(X)).
insert(person(X)) ==> insert(single(X)).
```

Example (Minimum with ECA rules in CHR)

```
insert(num(Y)), min(X) ==> Y<X | delete(min(X)).
num(Y), insert(min(X)) ==> Y<X | delete(min(X)).
insert(num(X)) ==> insert(min(X)).
```

LA formalism

- Hypothetical bottom-up logic programming language
- Features deletion of atoms and rule priorities
- Declarative production rule language, deductive database language, inference rules with deletion
- Designed to derive tight complexity results
- **The only implementation is in CHR**
- *It achieves the theoretically postulated complexity results!*

Logical Algorithm rules

Definition (LA rules)

$$r \,@\, p : A \rightarrow C$$

- r: rule name
- p: priority
  - arithmetic expression (variables must appear in first atom of A)
  - either dynamic (contains variables) or static
- A: conjunction of user-defined atoms and comparisons
- C: conjunction of user-defined atoms (variables must appear in A, i.e. range-restrictedness)
- $del(A)$: Deletion ("Negation") of positive atom A, overshadows A

Logical Algorithm semantics

Definition (LA semantics)

- *LA state*: set of user-defined atoms
  atoms occur positive, deleted (negative), or in both ways
- *LA initial state*: ground state
- Rule *applicable* to state if
  - lhs. atoms match state such that positive lhs. atoms do not occur deleted in state
  - lhs. comparisons hold under this matching
  - rhs. not contained in state (set-based semantics)
  - No other applicable rule with lower priority
- *LA final state*: no more rule applicable

Deletion by adding deletion atom `del`, no removal of atoms

Logical Algorithms in CHR

- Basically positive ground range-restricted CHR propagation rules
- Differences to CHR:
  - set-based semantics
  - explicit deletion atoms
  - redundancy test for rules to avoid trivial nontermination
  - rule priorities

Embedding LA in CHR

Definition (Rule scheme for LA predicate)

n-ary LA predicate a generates simpagation rules
($A = a(x_1, \ldots, x_n)$ with x_i distinct variables)

```
A \ A ⇔ true.
del(A) \ del(A) ⇔ true.
del(A) \ A ⇔ true.
```

Definition (Rule scheme for LA rule)

LA rule $r @ p : A \rightarrow C$ translates to CHR propagation rule with priority

$$r @ A_1 \Rightarrow A_2 \mid C : p$$

($A1$: atoms from A, $A2$: comparisons from A)

Priorities by CHR extension or conflict resolution

Ensuring set-based semantics

Applies to CHR rules in general (written as simplification rules)

- Generation of new rule variants by unifying head constraints

Definition (Rule scheme for set-based semantics)

To CHR simplification rules

$$H \wedge H_1 \wedge H_2 \Leftrightarrow G \mid B[\wedge H_1 \wedge H_2]$$

add rules (if guard does not imply that head is body)

$$H \wedge H_1 \Leftrightarrow H_1{=}H_2 \wedge G \mid B[\wedge H_1]$$

Example

```
a(1,Y), a(X,2) ==> b(X,Y).
```

Additional rule from unifying `a(1,Y)` and `a(X,2)`

```
a(1,2) ==> b(1,2).
```

LA example (Dijkstra's shortest paths)

Example (Dijkstra in LA)

```
d1 @   1: source(X) → dist(X,0)
d2 @   1: dist(X,N) ∧ dist(X,M) ∧ N<M → del(dist(X,M))
dn @ N+2: dist(X,N) ∧ edge(X,Y,M) → dist(Y,N+M)
```

Example (Dijkstra in CHR)

```
dist(X,N) \ dist(X,N) <=> true.
del(dist(X,N)) \ del(dist(X,N)) <=> true.
del(dist(X,N)) \ dist(X,N) <=> true.

d1 @ source(X) ==> dist(X,0) :1.
d2 @ dist(X,N), dist(X,M) ==> N<M | del(dist(X,M)) :1.
dn @ dist(X,N), edge(X,Y,M) ==> dist(Y,N+M) :N+2.
```

Set-based transformation does not introduce more rules

Deductive database languages (*)

Datalog

- Inference rules with negation similar to Prolog
- Negation as in production rule systems and Prolog
  - Only constants, no function symbols
  - Variables restricted to finite domains of constants
  - Rules are range-restricted
  - Stratification: No recursion through negation
- Evaluated bottom-up like Logical Algorithms
- Related to database language SQL

Constraint-based and logic-based programming

These are rule-based programming languages

- with logical variables subject to built-ins (like CHR)
- but no guards (except concurrent constraint languages)
- but no propagation rules (except deductive databases)
- but no multiple head atoms
- but with negation-as-failure and disjunction for search (in Prolog and constraint logic programming)

Prolog and Constraint Logic Programming

- Constraint logic programming (CLP) combines declarativity of logic programming and efficiency of constraint solving
- Prolog as CLP with syntactic equality as only built-in constraint
- Don't-know nondeterminism by choice of rule (or disjunct)
- (Don't-care nondeterminism by nonlogical cut operator)
- (Nonlogical Negation-as-failure)

Definition (CLP program)

CLP program: set of Horn clauses $A \leftarrow G$
(A atom, G conjunction of atoms and built-ins)

CHR with disjunction – $CHR^{\vee}$

- CHR with disjunction in body ($CHR^{\vee}$): declarative formulation and clear distinction between don't-know and don't-care nondeterminism
- Horn clause (CLP) program translates to equivalent $CHR^{\vee}$ program
- CLP head unification and clause choice moved to body of $CHR^{\vee}$ rule
- Required transformation is Clark's completion

Clark's completion

Definition (Rule scheme for Clark's completion for CLP clauses)

Clark's completion of predicate p/n defined by m clauses as

$$\bigwedge_{i=1}^{m} \forall(p(\bar{t}_i) \leftarrow G_i)$$

is the first-order logic formula

$$p(\bar{x}) \leftrightarrow \bigvee_{i=1}^{m} \exists \bar{y}_i \; (\bar{t}_i = \bar{x} \wedge G_i)$$

($\bar{t}_i$ sequences of n terms, $\bar{y}_i$ variables in G_i and t_i, $\bar{x}$ sequence of n new variables)

CLP translation to CHR

For pure Prolog and CLP without cut and negation-as-failure

Definition (Rule scheme for pure (C)LP clauses)

- CLP predicate p/n is considered as CHR constraint
- For each predicate p/n Clark's completion of p/n added as $CHR^{\vee}$ simplification rule

Example (Append in Prolog)

```
append([],L,L) ← true.
append([H|L1],L2,[H|L3]) ← append(L1,L2,L3).
```

Example (Append in $CHR^{\vee}$)

```
append(X,Y,Z) ⇔
    (    X=[]∧Y=L∧Z=L
    ∨    X=[H|L1]∧Y=L2∧Z=[H|L3]∧append(L1,L2,L3) ).
```

Example - Prime sieve programs

Comparison between Prolog and CHR by example program

Example (Prime sieve in Prolog)

```
primes(N,Ps):- upto(2,N,Ns), sift(Ns,Ps).

upto(F,T,[]):- F>T, !.
upto(F,T,[F|Ns1]):- F1 is F+1, upto(F1,T,Ns1).

sift([],[]).
sift([P|Ns],[P|Ps1]):- filter(Ns,P,Ns1), sift(Ns1,Ps1).
filter([],P,[]).

filter([X|In],P,Out):- X mod P =:= 0, !, filter(In,P,Out).
filter([X|In],P,[X|Out1]):- filter(In,P,Out1).
```

Prolog uses nonlogical cut operator.

Example (Prime sieve in CHR)

```
       upto(N) <=> N>1 | M is N-1, upto(M), prime(N).
sift @ prime(I) \ prime(J) <=> J mod I =:= 0 | true.
```

Example - Shortest path program

Comparison between Prolog and CHR by example program

Example (Shortest path in Prolog)

```
p(From,To,Path,N) :- e(From,To,N).
p(From,To,Path,N) :- e(From,Via,1),
                     not member(Via,Path),
                     p(Via,To,[Via|Path],N1),
                     N is N1+1.
shortestp(From,To,N) :- p(From,To,[],N),
                     not (p(From,To,[],N1),N1<N).
```

Prolog uses nonlogical negation-as-failure.

Example (Shortest path in CHR)

```
p(X,Y,N) \ p(X,Y,M) <=> N=<M | true.
e(X,Y) ==> p(X,Y,1).
e(X,Y), p(Y,Z,N) ==> p(X,Z,N+1).
```

Concurrent constraint programming

- Concurrent constraint (CC) language framework
  - Permits both nondeterminisms
  - One of the frameworks closest to CHR
  - We concentrate on the committed-choice fragment of CC (Based on don't-care nondeterminism like CHR)

Definition (Abstract syntax of CC program)

CC program is a finite sequence of declarations that define agent.

$$
\begin{array}{lll}
\text{Declarations} & D ::= & p(\tilde{t}) \leftarrow A \mid D, D \\
\text{Agents} & A ::= & \mathit{true} \mid c \mid \sum_{i=1}^{n} c_i \rightarrow A_i \mid A \| A \mid p(\tilde{t})
\end{array}
$$

(p user-defined predicate symbol, $\tilde{t}$ sequence of terms, c and c_i's constraints)

Ask-and-tell

- **Ask-and-tell**: communication mechanism of CC (and CHR)
- **Tell**: Add a constraint to the constraint store (producer / server)
- **Ask**: Inquiry whether or not constraint holds (consumer / client)
  - Realized by logical entailment
  - Checks whether constraint is implied by constraint store
- Generalizes idea of concurrent data flow computations
  - Operation waits until its parameters are known

CC operational semantics (I)

States are pairs of agents and built-in constraint store

Definition (Ask and Tell)

Tell: adds constraint c to constraint store

$$\langle c, d\rangle \rightarrow \langle true, c \wedge d\rangle$$

Ask: nondeterministically choose constraint c_i (implied by d) and continue with agent A_i

$$\langle \sum_{i=1}^{n} c_i \rightarrow A_i, d\rangle \rightarrow \langle A_j, d\rangle \quad \text{if} \quad CT \models \forall(d \rightarrow c_j)\ (1 \leq j \leq n)$$

CC operational semantics (II)

Definition (Composition and Unfold)

Composition: Operator $\parallel$ defines concurrent composition of agents

$$\frac{\langle A, c\rangle \rightarrow \langle A', c'\rangle}{\begin{array}{c}\langle (A \parallel B), c\rangle \rightarrow \langle (A' \parallel B), c'\rangle \\ \langle (B \parallel A), c\rangle \rightarrow \langle (B \parallel A'), c'\rangle\end{array}}$$

Unfold: replaces agent $p(\tilde{t})$ by its definition

$$\langle p(\tilde{t}), c\rangle \rightarrow \langle A, \tilde{t} = \tilde{s} \wedge c\rangle \quad \text{if } (p(\tilde{s}) \leftarrow A) \text{ in program } P$$

Embedding in CHR

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

Translation

Definition (Rule scheme for CC expressions)

Declarations and agents are translated from CC

$$\begin{aligned} D &::= \quad p(\tilde{t}) \leftarrow A \mid D, D \\ A &::= \quad true \mid c \mid \textstyle\sum_{i=1}^{n} c_i \rightarrow A_i \mid A \| A \mid p(\tilde{t}) \end{aligned}$$

to CHR as

$$\begin{aligned} D^{CHR} &::= \quad p(\tilde{t}) \Leftrightarrow A \mid D, D \\ A^{CHR} &::= \quad true \mid c \mid \mathtt{ask}(\textstyle\sum_{i=1}^{n} c_i \rightarrow A_i) \mid A \wedge A \mid p(\tilde{t}) \end{aligned}$$

For each CC Ask A of the form $\sum_{i=1}^{n} c_i \rightarrow A_i$ also generate n single-headed simplification rules for unary `ask` constraint

$$\mathtt{ask}(A) \Leftrightarrow c_i \,|\, A_i \ (1 \leq i \leq n).$$

Example (Maximum)

Example (Maximum in CC)

```
max(X,Y,Z) ← (X≤Y → Y=Z) + (Y≤X → X=Z)
```

Example (Maximum in CHR)

```
max(X,Y,Z) ⇔ ask((X≤Y → Y=Z) + (Y≤X → X=Z)).

ask((X≤Y → Y=Z) + (Y≤X → X=Z)) ⇔ X≤Y | Y=Z.
ask((X≤Y → Y=Z) + (Y≤X → X=Z)) ⇔ Y≤X | X=Z.
```

To simplify rules replace `ask((X≤Y → Y=Z) + (Y≤X → X=Z))` by `ask_max(X,Y,Z)`

Example (Simplified maximum in CHR)

```
ask_max(X,Y,Z) ⇔ X≤Y | Y=Z.
ask_max(X,Y,Z) ⇔ Y≤X | X=Z.
```

Embeddings in CHR

Advantages

- Advantages of CHR for **execution**
  - Efficiency, also optimal complexity possible
  - Abstract execution by constraints, even when arguments unknown
  - Incremental, anytime, online algorithms for free
  - Concurrent, parallel for confluent programs
- Advantages of CHR for **analysis**
  - Decidable confluence and operational equivalence
  - Estimating complexity semi-automatically
  - Logic-based declarative semantics for correctness
- *Embedding allows for comparison and cross-fertilization (transfer of ideas)*

Potential shortcomings of embeddings in CHR

⇒ Use extensions of CHR (dynamic CHR covers all

- for built-in “**negation**” of rb systems, deductive db and Prolog
  ⇒ CHR with negation-as-absence
- for **conflict resolution** of rule-based systems
  ⇒ CHR with priorities
- for built-in **search** of Prolog, constraint logic programming
  ⇒ CHR with disjunction or search library
- for ignorance of **duplicates** of rule-based formalisms
  ⇒ CHR with set-based semantics
- for **diagrammatic notation** of graph-based systems
  ⇒ CHR with graphical interface

Instead of extensions, special-purpose CHR programs can be used.

Positive ground range-restricted CHR

- All approaches can be embedded into simple CHR fragment (except Prolog, constraint logic programming)
  - **ground**: queries ground
  - **positive**: no built-ins in body of rule
  - **range-restricted**: variables in guard and body also in head
- These conditions imply
  - Every state in a computation is ground
  - CHR constraints do not delay and wake up
  - Guard entailment check is just test
  - Computations cannot fail
- Conditions can be relaxed: auxiliary functions as non-failing built-ins in body

Distinguishing features of CHR for programming

Unique combination of features

- **Multiple Head Atoms** not in other programming languages
- **Propagation rules** only in deductive db, Logical Algorithms
- **Constraints** only in constraint-based programming
  - Logical variables instead of ground representation
  - Constraints are reconsidered when new information arrives
  - Notion of failure due to built-in constraints
- **Logical Declarative Semantics** only in logic-based prog.
  - CHR computations justified by logic reading of program

Embedding fragments of CHR in other rule-based approaches

Possibilities are rather limited (without interpreter or compiler)

- **Positive ground range-restricted fragment** embeddable into
  - Rule-based systems with negation and Logical Algorithms
  - Only simplification rules in Rewriting- and Graph-based approaches (except Petri-nets)
  - Only propagation rules in deductive databases
- **Single-headed rules** embeddable into
  - Concurrent constraint programming languages

The Potential of Constraint Handling Rules

CHR - an essential unifying computational formalism?
Rule-based Systems, Formalisms and Languages can be compared and cross-fertilize each other via CHR!

- CHR is a logic *and* a programming language
- CHR can express any algorithm with optimal complexity
- CHR is efficient and extremly fast
- CHR supports reasoning and program analysis
- CHR programs are anytime, online and concurrent algorithms
- CHR has many applications from academia to industry

The first formalism and the first language for students
Reasoning formalism and programming language for research
CHR - a Lingua Franca for computer science!