ulm university universität ulm

Rule-Based Programming

Based on book Constraint Handling Rules

Prof. Dr. Thom Frühwirth | 2021 |
University of Ulm, Germany

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

Embedding rule-based approaches in CHR

Using source-to-source transformation for embeddings

- Rewriting- and graph-based **formalisms**
  - GAMMA Multiset Transformation
  - Term Rewriting Systems and Functional Programming (also a language)
  - Colored Petri Nets
- Rule-based **systems**
  - Production Rules
  - Event-Condition-Action Rules
  - Logical Algorithms (also a formalism)
- Logic- and constraint-based **programming languages**
  - Datalog for Deductive Databases (also a system)
  - Prolog and Constraint Logic Programming
  - Concurrent Constraint Programming

Advantages of Embeddings in CHR

Advantages of CHR for **execution**

- Effective and efficient high-level embeddings
- Abstract (non-ground) execution for compile-time partial evaluation and program transformation
- Incremental, anytime, online algorithms for free
- Concurrent, parallel for confluent programs
- Completion can make programs confluent

Advantages of CHR for **analysis**

- Decidable confluence and operational equivalence
- Estimating complexity semi-automatically
- Logic-based declarative semantics for correctness

Embeddings for comparison and cross-fertilization (transfer of ideas)

Positive ground range-restricted CHR

- Approaches can be embedded into simple CHR fragment (except constraint logic programming and concurrent programming)
  - **positive**: no built-ins in body of rule
  - **ground**: queries ground
  - **range-restricted**: variables in body also occur in head
    or as result of auxiliary functions as non-failing built-ins in body
- These conditions imply
  - Every state in a computation is ground
  - CHR constraints need to do not wake up
  - Guard entailment check is just test
  - Computations cannot fail

Rewriting-based and graph-based formalisms (I)

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

Rewriting-based and graph-based formalisms (II)

- 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 in CHR

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)**

GAMMA

- Chemical metaphor: molecules in solution react according to reaction rules (transitions)
- Reaction in parallel on disjoint sets of molecules (states)
- Basis for Chemical Abstract Machine (ChAM)

Definition (GAMMA)

- GAMMA program: pairs $(c/n, f/n)$ (predicate c, function f)
- f applied to molecules in solution S for which c holds
- Result $f(x_1, \ldots, x_n) = \{y_1, \ldots, y_m\}$ replaces $\{x_1, \ldots, x_n\}$ in S
- Repeat until exhaustion (no more application possible)

GAMMA Translation

- Molecules modeled as unary CHR constraints, reactions as rules

Definition (Rule scheme for GAMMA pair)

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

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

where $d/1$ wraps molecules, c/n is built-in, f/n is defined by rules

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

where C guard and D are auxiliary constraints.

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).
```

Compile-Time Optimization of CHR - Partial Evaluation with Rule Removal

Definition (Partial Evaluation with Rule Removal)

- Given a terminating and confluent (well-behaved) CHR program with a generalized simpagation rule

  `H1 \ H2 ⇔ C | B`

- $\mapsto^*$ is the reflexive transitive closure of the transition relation $\mapsto$.
- **Execute** `H1,C,B` $\mapsto^*$ `E`. **Replace** rule body by the answer and simplify rule into:

  `H0 \ H1-H0, H2 ⇔ C | E-H0-C`

  `H0` are the common CHR constraints in `H1` and `E`.
  `G-H` are the constraints `G` without the constraints `H`.
- An *internal constraint* never occurs in a query (initial state).
- If an internal constraint is not used anymore, **remove** the rules that define it.

Partial Evalution - GAMMA Example

Example (Greatest Common Divisor)

Terminating and confluent CHR program.

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

Execute `X<Y, gcdsub(X,Y)` $\mapsto^*$ `X<Y, d(X), d(Y-X).`
Replace rule body by answer. Make it a simpagation rule.
Remove rules defining internal constraint `gcdsub`.

```
gcd @ d(X) \ d(Y) <=> X<Y | d(Y-X).
```

This is just the `gcd` rule of the CHR introduction.

Why not partially evaluate the rule for `gcdsub`?

GAMMA with γ-Abstraction - Translation

γ-abstraction introduces explicit variable arguments. It allows for nameless functions and conditions regarded as boolean functions.

Definition (Rule scheme for GAMMA pair with variables)

GAMMA pair with γ-abstraction

$g = (\gamma(x_1, \ldots, x_n) : Cond, (f_1(\ldots), \ldots, f_m(\ldots)))$

where $Cond$ is a condition on the molecules $x_1, \ldots, x_n$

and $f_1, \ldots, f_m$ are functions applied to these molecules,

is translated to simplification rule

$$g \mathbin{@} d(x_1), \ldots, d(x_n) \Leftrightarrow Cond \mid f_1(\ldots, y_1), \ldots, f_m(\ldots, y_m),\ d(y_1), \ldots, d(y_m)$$

where the f_i are built-in or defined by rules of the form

$$f_i(\ldots, y_i) \Leftrightarrow G \mid D$$

GAMMA with γ-Abstraction - Examples

Example (Minimum)

```
min = ([X,Y]:X<Y,[X])
min @ d(X), d(Y) <=> X<Y | d(X).
```

Example (Greatest Common Divisor)

```
gcd = ([X,Y]:X<Y,[X,Y-X])
gcd @ d(X), d(Y) <=> X<Y | d(X), Z is Y-X, d(Z).
```

Example (Prime sieve)

```
prime = ([X,Y]:X div Y,[X])
prime @ d(X), d(Y) <=> X div Y | d(X).
```

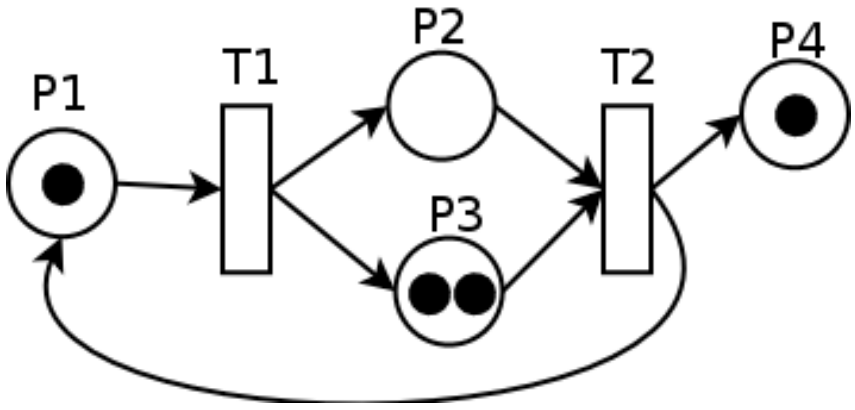
Petri Nets

Diagrammatic model for concurrent distributed systems, for nondeterministic discrete-event dynamic systems.

- Petri Net is a directed bipartite graph with
  - Place nodes (P) ($\circ$)
  - Transition nodes (T) ($\square$)
  - Arcs ($\rightarrow$)
  - Tokens ($\bullet$)
- Tokens reside in places, move along arcs through transitions
- Transitions fire if there are tokens in places of all incoming arcs:
  - One token removed from place for each incoming arc, one token placed in place of each outgoing arc

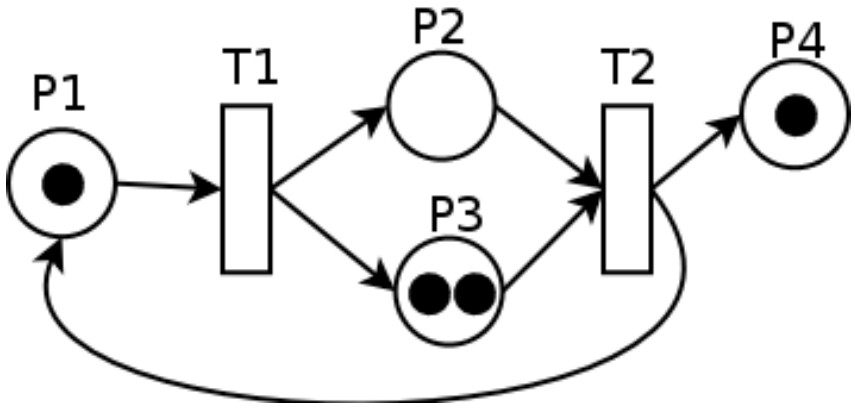
Petri Nets often describe nonterminating processes. Still, many properties are decidable, because Petri Nets are not Turing-complete.

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

Example (Petri Net) in CHR



Standard Petri Nets translate to tiny fragment of CHR

- Nullary CHR constraints for places
- Simplification rules without guards and without built-in constraints

```
t1 @ p1 ⇔ p2, p3
t2 @ p2, p3 ⇔ p1, p4
```

Colored Petri Nets (CPN)

Tokens have different colors (values)

- Places constrained to 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 still not Turing-complete.

Translation

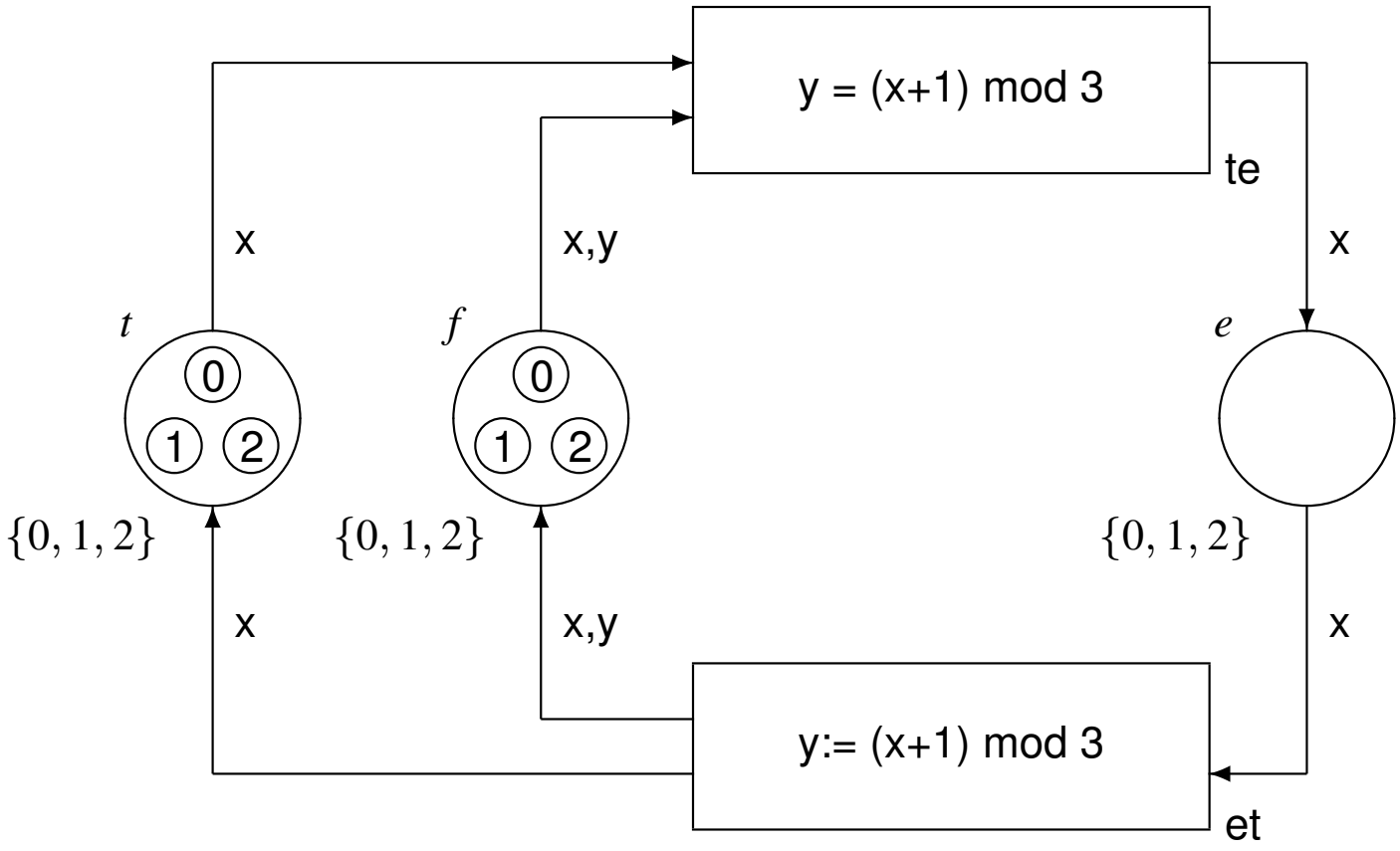
Simplification rules over unary CHR constraints and finite data

- Places $\rightarrow$ unary CHR constraint symbols for tokens
- Tokens $\rightarrow$ arguments of place constraints
- Transitions $\rightarrow$ CHR simplification rules
  - Places of incoming arcs $\rightarrow$ rule head
  - Transition guard $\rightarrow$ rule guard
  - Transition equation $\rightarrow$ rule body
  - Places of outgoing arcs $\rightarrow$ rule body

Example – Dining Philosophers

- Classical Fairness problem: Competition for limited ressources
  - Philosophers sit at round table, between each philosopher one fork
  - Philosophers either eat or think
  - For eating, forks from both sides required
  - After eating for a while, philosophers start thinking again
  - Nonterminating system
- Three Dining Philosophers as Colored Petri Net
  - Philosopher as colored token (integer)
  - Tokens x and y are neighbors iff $y = (x + 1) \mod 3$
  - Places: eat `e`, think `t`, fork `f`
  - Transitions: `et` and `te`

Three dining philosophers Petri Net



Three dining philosophers CHR translation

Example (Dining philosophers in CHR)

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

- Note: `et` rule is reverse of `te` rule
- Execution in CHR is not fair [but see Conflict Resolution]
- CPN fixes number of philosophers
- CHR more general, number can be a parameter

Examples of cross-fertilization by CHR embedding

- State Reachability Analysis of CPN applicable to subset of CHR
- Confluence Analysis of CHR applicable to CPN for fairness

Comparison: GAMMA, Colored Petri Nets, PGRS-Subset of CHR

GAMMA pair with γ-abstraction in CHR (only unary constraint $d/1$):

$$g \mathbin{@} d(x_1), \ldots, d(x_n) \Leftrightarrow \mathit{Cond} \mid f_1(\ldots, y_1), \ldots, f_m(\ldots, y_m),\ d(y_1), \ldots, d(y_m)$$

Colored Petri Net in CHR (only unary constraints, finite data types):

$$t \mathbin{@} p_1(x_1), \ldots, p_n(x_n) \Leftrightarrow \mathit{Cond} \mid f_1(\ldots, y_1), \ldots, f_m(\ldots, y_m),\ p'_1(y_1), \ldots, p'_m(y_m)$$

Positive Ground Range-Restricted Simplification Rules of CHR:

$$r \mathbin{@} c_1(\bar{x}_1), \ldots, c_n(\bar{x}_n) \Leftrightarrow \mathit{Cond} \mid \mathit{BuiltIns},\ c'_1(\bar{y}_1), \ldots, c'_m(\bar{y}_m)$$

GAMMA can embed CHR constraints as structured data $d(c_i(\bar{x}_i))$.
Petri Nets only if data is finite as token in place $c_i(t(\bar{x}_i))$.
Can analyse and draw GAMMA and PGRS-CHR as Petri Nets.

Term Rewriting Systems (TRS)

Definition (Operational Semantics of TRS)

A TRS is a set of rewrite (TRS) rules $S \to T$ (S, T are terms).
Rule application until exhaustion:

$$U[S\beta]_p \to U[T\beta]_p \text{ if } S \to T$$

Given a term U with a sub-term V at position p that matches S with substitution β. Then we replace V by T under substitution β.

Definition (Declarative Semantics of TRS)

A TRS rule $S \to T$ is interpreted as axiom formula $\forall(S = T)$, where $=$ is the syntactic equality relation over terms of first-order logic.
Relation $=$ is reflexive, symmetric, transitive, and ($\doteq$ is identity)
$f(X_1,\ldots,X_n) = g(Y_1,\ldots,Y_m) \Leftrightarrow f \doteq g \wedge n \doteq m \wedge X_1 = Y_1 \wedge \ldots \wedge X_n = Y_m$

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 multiple occurrences of variables

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

Embedding TRS in CHR

Embedding in CHR uses equality from declarative semantics of TRS

Definition (Rule scheme for term rewriting rule)

TRS rule

$$S \to T$$

translates to CHR simplification rule

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

(X new variable, $\mathtt{eq}$ CHR constraint)

Square brackts $[\ldots]$ denote flattening function (see next slide)

Flattening

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

- Opposite of variable elimination, introduce new variables
- Flattening function transforms atomic equality constraint $\mathtt{eq}$ 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)

Size grows by linear factor. Generates tree structure with equations as nodes.

Example (Addition of natural numbers)

Successor notation for natural numbers: `s(N)` = $N + 1$.

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.
```

CHR translation also works for non-ground queries, TRS only for ground queries.

Example (Logical conjunction)

Boolean function over `0` for *false* and `1` for *true*.

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.
```

CHR translation also works for non-ground queries, TRS only for ground queries.

Completeness of Embedding and Nonlinearity of TRS

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

Example

In TRS, rule `and(X,X) -> X` applicable to

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

In CHR embedding, rule

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

not directly applicable to query

```
X eq and(X1,X2), X1 eq and(X3,X4), X3 eq 0, X4 eq 1,
X2 eq and(X5,X6), X5 eq 0, X6 eq 1
```

because `and(X3,X4)` and `and(X5,X6)` are different.

Completene Embedding for nonlinearity TRS

Definition (Rule scheme variant for nonlinear TRS)

For nonlinear lhs: Propagate syntactic equality upwards in tree.

$$X \texttt{ eq } T \wedge Y \texttt{ eq } T \Rightarrow X = Y$$

For nonlinear rhs: Keep equations except for root variable.

$$\bigwedge_{i=1}^{n} [X_i \texttt{ eq } T_i] \quad \backslash \quad X \texttt{ eq } f(X_1, \ldots, X_n) \Leftrightarrow X \texttt{ eq } g(Y_1, \ldots, Y_m) \wedge \ldots$$

Example (Boolean conjunction)

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

Associative-Commutative (AC) Term Rewriting

Example (Associative-Commutative Addition)

```
commutativity: X+Y -> Y+X
associativity: X+(Y+Z) -> (X+Y)+Z
```

But first rule does not terminate. Lack of unique normal form.

Low-level implementation in TRS. In CHR, exploit multiset semantics.

Definition (Associative-Commutative Transformation)

Declare binary function `F` to be AC with built-in `ac(F)`.

Commutativity

```
T eq T0 ⇔ T0=..[F,T1,T2],ac(F) | T has F:T1, T has F:T2.
```

Associativity

```
% T has F:T1 \ T1 has F:T2 ⇔ ac(F) | T has F:T2.
T1 has F:T2 \ T has F:T1 ⇔ ac(F) | T=T1.
T has F:T1 \ T eq T2 ⇔ ac(F) | T has F:T0, T0 eq T2.
```

Assoc. incompatible with equality propagation, since `T1` becomes `T`.

Compile-Time Optimization of CHR - Partial Evaluation for Rule Heads

Definition (Partial Evaluation for Rule Heads)

- Given a terminating and confluent (well-behaved) CHR program with a generalized simpagation rule

  `r @ H1 \ H2 ⇔ C | B`

- **Execute** `H1,H2,C` $\mapsto^*$ `D,E` (`D` built-ins, `E` CHR constraints) without using rule `r`.
- **Replace** rule head and guard by answer and simplify rule:

  `r' @ H0 \ E-H0 ⇔ D | B-H0-D`

  `H0` are the common CHR constraints in `E` and `B`.

CHR Compile-Time Program Transformation

The transformation is defined by CHR rules. Check correctness.

Definition (CHR Program Transformation)

- Given: well-behaved transformation rules, terminating program
- *For each* generalized simpagation rule from the original program

  `r @ H1 \ H2 ⇔ C | B`
- **Execute** `H1,H2,C` $\mapsto^*$ `D,E` (`D` built-ins, `E` CHR constraints) with the transformation rules only.
- **Replace** rule head and guard by answer and simplify rule:

  `r' @ H0 \ E-H0 ⇔ D | B-H0-D`

  `H0` are the common CHR constraints in `E` and `B`.
- Apply compile-time partial evaluation to the rule bodies.
  If original program not confluent, apply only transformation rules.

Example (AC-Addition of Natural Numbers)

Example (TRS with low level implemention)

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

Example (CHR standard embedding of TRS)

```
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 (CHR with AC of addition)

```
T eq T0 <=> T0=..[F,T1,T2],ac(F) | T has F:T1,T has F:T2.
T has (+):T1,T has (+):T2,T1 eq 0,T2 eq Y <=> T eq Y.
T has (+):T1,T has (+):T2,T1 eq s(T3),T3 eq X,T2 eq Y <=>
   T eq s(T4),T4 has (+):T5,T4 has (+):T6,T5 eq X,T6 eq Y.
```

Functional Programming (FP)

FP can be seen as programming language based on TRS formalism

- Extended by built-in functions and guard tests
- Distinguish between functions (defined by rules) and data
- Syntactic restriction on lhs (left-hand-side) of rewrite rule:
  No nesting of functions, functions defined over data only
  (Exceptions possible, e.g. higher-order functions)

Translation

Definition (Rule scheme for functional program rule)

FP rewrite rule

$$S \to G \mid T$$

translates to CHR simplification rule

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

(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.
```

Generic rules applied at compile-time by partial evaluation of CHR

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

Higher Order Functions in FP

Higher order functions have functions as arguments

Example (Higher Order Function twice)

Apply any unary function twice on its argument

`twice(F,A) --> F(F(A))` (`F`: function, `A`: argument)

So given `inc(X) --> X + 1`

then `twice(inc,7) --> inc(inc(7)) --> 9`

Higher Order Functions in CHR

Use auxiliary function: `apply(F,[A1,...,An])`

applies function `F` to the arguments `A1,...An`

Example (Higher Order Function twice)

Function `twice` is defined as:

```
twice(F,A) --> apply(F,[apply(F,[A])])
```

Extend translation of FP by a definition of `apply`

Definition (Transformation Rule for apply/2)

```
X eq apply(F,Args) <=>
  atom(F),
  is_list(Args) |
    G =.. [F|Args],
    X eq G.
```

Higher Order Function twice

Example: twice – FP definition

```
twice(F,A) --> apply(F,[apply(F,[A])])
```

Example: twice – CHR embedding with transformation

```
X eq twice(F,A) <=>
     X eq apply(F1,[X1]),
     F1 eq F, F1 eq F,
     X1 eq apply(F1,[A1]),
     A1 eq A.
```

Example: twice – query

```
?- evaluate(X eq twice(inc,6)).
 X = 8 .
?- evaluate(X eq twice(fib,6)).
 X = 377 .
```

Higher Order List Processing Functions

Example: map – FP definition

```
map(F,[]) --> [].
map(F,[X|Xs]) --> [apply(F,[X])|map(F,Xs)].
```

Example: map – query

```
?- evaluate(X eq map(fib,[1,2,3,4,5])).
 X = [1, 2, 3, 5, 8] .
```

Example: foldr – FP definition

```
foldr(F,Z,[]) --> Z.
foldr(F,Z,[X|Xs]) --> apply(F,[X,foldr(F,Z,Xs)]).
```

Example: foldr – query

```
?- evaluate(X eq foldr(+,0,[1,2,3,4,5,6,7,8,9])).
 X = 45 .
```

λ Abstraction for Anonymous Functions

An anonymous function does not have a name. It consists of a paramter list and its definition. (Like γ abstraction in GAMMA.) Useful in higher order programming. We implement it with apply/2.

Definition (Transformation Rule for apply/2 with λ abstraction)

```
X eq apply(L:E,Args) <=>
  is_list(L), is_list(Args) |
  copy_term(L:E,Args:EC),
  X eq EC.
```

`copy_term` copies a term using new variables.
So variables in the λ abstraction remain unbound.

λ Abstraction Examples

Examples

```
?- evaluate(X eq twice([Y]:Y*Y,7)-1).
 X = 2400 .

?- evaluate(X eq map([Y]:Y*Y,map([X]:X-1,[2,3,4,5,6]))).
 X = [1, 4, 9, 16, 25] .

?- evaluate( X eq foldr([U,V]:U*V+fib(U),1,[1,2,3,4])).
 X = 63 .
```

Comparison: TRS and FP with AC, PGRS-Subset of CHR

- TRS lacks **built-ins**. FP adds built-ins and data.
- FP lacks for definitions of **higher-order** (nested) functions. They are predefined (builtin).
- TRS and FP cannot directly express **AC (associativity and commutativity)** of functions. It can be builtin in TRS.

PGRS-CHR program in FP under two conditions:

- Each CHR constraint can be rewritten as function: make one argument its result (always a variable), e.g. `add(X,Y,Z)` becomes `Z eq add(X,Y)`.
- If the `eq` constraints of the resulting rewritten rules form a tree, then unflattening into FP rule possible.

PGRS-Subset of CHR in FP extended by AC

With AC transfered from TRS to FP, define a multi-set union operator for conjunction of CHR constraints interpreted as Boolean functions.

Example (PGRS-CHR in FP with AC of `u`)

```
% maximum multiset transformation
max(A) u max(B) --> A=<B | max(B).

?- evaluate(X eq max(3) u max(7) u max(5)).
X = max(7) .

% destructive assignment
assign(V,N) u cell(V,O) --> cell(V,N).

?- evaluate(X eq cell(x,5) u assign(x,3) u assign(x,7)).
X = cell(x, 7) .
```

Rule-based systems (I)

For implementation of practical applications

- **Production rule systems**
  - First rule-based systems
  - No declarative semantics
  - Developed in the 1980s
- **Event-Condition-Action (ECA) rules**
  - Extension of production rules for active database systems
  - Rules react to updates and other events
  - Hot research topic in the 1990s
  - Some aspects standardized in SQL-3

Rule-based systems (II)

- **Business rules in Workflow Systems**
  - 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**
  - Hypothetical declarative production rule language
  - Permanent deletion by overshadowing information
  - More recent approach (early 2000s)
  - Also a rule-based formalism
- **Datalog**
  - Deductive-database query language
  - Subset of Logical Algorithms (no deletion)
  - Subset of logic programming language Prolog (only finite data)
  - Developed in the late 1980s, research topic again now

Production rule systems

Working memory stores facts (working memory elements, WME)
Facts have name and named attributes (like records or objects)

Production rule

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

- **If**-clause: *Condition*
  - Expression matchings describing facts (patterns)
- **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 applicable 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 in CHR with *refined semantics* or CHR language 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 `-(PositivePattern)` in production rules

- Satisfied if no fact satisfies condition
- Facts correspond to data (not operations)

Example (Minimum in OPS5)

A number is the minimum if there is no other number that is smaller.

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

Compact, but inefficient and non-monotonic...

Non-Monotonicity of Logic Formalisms and Rule-Based Systems

Monotonicity:

- Logic: Adding formulae to a theory allows to draw more conclusions (logical consequences, inferences), e.g. classical first-order predicate logic.
- Rule-Based System: Adding rules to program or constraints to initial state allows more rule applications, e.g. CHR abstract semantics.

Nonmonotonicity (Defeasible Reasoning):

Maybe closer to human commonsense reasoning.

- Logic: Conclusion may be retracted if formula is added.
- Rule-Based System: Rule application not possible anymore if information is added.

There may be conflicting rule applications $\Rightarrow$ **Conflict Resolution**.

Negation by Closed-World-Assumption (CWA)

- **Classical negation** in first-order logic is monotonic.
- In Prolog, **negation-as-failure** is non-monotonic: If we cannot derive something, we conclude its negation holds.
- In production rules, **negation-as-absence** is non-monotonic: If a fact is not found, we conclude its negation holds.

Example (Minimum in OPS5, contd.)

Non-monotonic because it uses negation-as-absence.
Given a set of number facts, the rule computes the minimum.
But if we add smaller number in the beginning, the minimum changes.

If we add the new number later on, we get a second minimum.
So not confluent.

Negation-as-Absence to Restrict Rule Applicability

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>) )
```

Negation-as-absence is used to avoid non-termination and duplicate facts.

Reasoning with Default and Exceptions

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

Example (Marital Status in OPS5)

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

Status `single` is default if not married (as opposed to `divorced`).
What happens if person marries? What happens if person divorces?

Translation of Negation-as-Absence into CHR

- Two approaches and one special case
  - Built-in `find_chr_constraint` in guard (low level)
  - CHR constraint to check for absence (requires refined semantics, easier to analyse)
  - Special Case: Repair - retract obsolete conclusions (changes semantics)
- Assume w.l.o.g. one negation per rule (not nested)
- Positive rule parts translated as before
- Transfer to CHR for arbitrary generalised simpagation rules

CHR rules with negation-as-absence

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

CHR generalised simpagation rule with unique identifier `N` and negated pattern (`NH` CHR constraints, `NC` built-in constraints)

`N @ H1 \ H2 -(NH,NC) ⇔ C | B`

translates to CHR rules

`N1 @ fire ∧ H1 ∧ H2 ⇒ C | check(N,`$vars$`(H1,H2,C))`
`N2 @ NH \ check(N,`$vars$`(H1,H2,C)) ⇔ NC | true`
`N3 @ H1 \ H2 ∧ check(N,`$vars$`(H1,H2,C)) ⇔ B`

- $vars(\bar{t})$ denotes the list of variables of its arguments $\bar{t}$
- **phase constraint** `fire/0` triggers rules, is removed at end
- `check/2` checks for absence of negated pattern

CHR rules with negation-as-absence (II)

Explanation

```
N1 @ fire ∧ H1 ∧ H2 ⇒ C | check(N,vars(H1,H2,C))
N2 @ NH \ check(N,vars(H1,H2,C)) ⇔ NC | true
N3 @ H1 \ H2 ∧ check(N,vars(H1,H2,C)) ⇔ B
```

- `fire,H1,H2` found, `check` for absence of `NH` using rule `N1`
- **If** `NH` found using rule `N2`, **then** remove `check`, nothing done
- **Else** apply original rule using `N3`, remove `H2` and `check`, add `B`
- Relies on rule order between `N2` and `N3`
- Works under **refined semantics**

Example Minimum with Negation-as-Absence

Example (Translation Minimum in CHR)

```
% min @ num(X), -(num(Y),Y<X) ==> min(X).
fire, num(X) ==> check(min,[X]).
num(Y) \ check(min,[X]) <=> Y<X | true.
num(X) \ check(min,[X]) <=> min(X).
```

Example (Minimum Queries)

```
?- num(2),num(1),num(3).          % code without phase constraint
   num(2),num(1),num(3), min(2),min(1)  % not correct

?- fire, num(2),num(1),num(3).    % fire at start
   num(2),num(1),num(3)           % fire should come at end

?- num(2),num(1),num(3),fire.     % fire at end
   num(2),num(1),num(3), min(1) % minimum correctly computed

?- num(2),num(1),num(3),fire, num(0),num(4),fire. % fire twice
   num(2),...,num(4), min(1), min(0) % incorrect, not incremental
```

More Examples

Example (OPS5 Transitive closure in CHR, init-path rule)

```
% path @ e(X,Y), -(p(X,Y),true) ==> p(X,Y).

fire, e(X,Y) ==> check(path,[X,Y]).
p(X,Y) \ check(path,[X,Y]) <=> true.
e(X,Y) \ check(path,[X,Y]) <=> p(X,Y).
```

Example (OPS5 Marital status in CHR)

```
% status@person(X),-(married(X),true) ==> single(X).

fire, person(X) ==> check(status,[X]).
married(X) \ check(status,[X]) <=> true.
person(X)  \ check(status,[X]) <=> single(X).
```

CHR propagation rules with special-case negation-as-absence

Assume negative pattern holds, otherwise repair by undoing body

- If `B` has no built-ins, contains all head variables:
- Then use `B` directly instead of auxiliary `check`
- Changes semantics, hence may not be correct

Definition (Rule scheme for CHR propagation rule with negation)

CHR propagation rule

```
N @ H1 -(NH,NC) ⇒ C | B
```

translates to CHR rules

```
N2 @ NH \ B ⇔ NC | true   % repair by removing B
N' @ H1 ⇒ C | B           % ignore negated pattern
```

- Rules are ordered: `N2` rules have to come before `N'` rules

Consequences and Examples

- Shorter, more concise programs, often confluent, incremental, concurrent, declarative ⇒ better execution and analysis
- ⇒ Negation may not be 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).
```

Conflict Resolution

Definition (Conflict Resolution)

Choose rule to be applied among applicable rules according to given strategy based on property of rules.

Definition (Conflict Resolution by Weight)

- Give rules a **weight (priority, preference)**.
- Weights known at compile-time (static) or computed at run-time (dynamic).
- Choose rule with highest weight.
- Requires (total) order on weights.

Incremental 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 ⇔ C | B : P
```

translates to CHR rules

```
find @ H1 ∧ H2 ⇒ C | conflictset([rule(P,Id,vars(H1,H2,C))])
apply @ H1 \ H2 ∧ apply(rule(P,Id,vars(H1,H2,C))) ⇔ B
```

- Rule `find`: finds applicable rules
- Constraint `conflictset/1`: collects applicable rules
- Rule `apply`: executes chosen rule given in `apply/1`

Incremental conflict resolution (II)

Additional generic rules for rule choice

Rules to resolve conflict

```
collect @ conflictset([R]) ∧ conflictset(L) ⇔ conflictset([R|L])
choose  @ fire ∧ conflictset(L) ⇔ L=[_|_] |
            choose(L,R,L1) ∧ conflictset(L1) ∧ apply(R) ∧ fire
```

Plus rules for clean-up and garbage-collection of auxiliary constraints

- Rule `collect` collects applicable rules into one `conflictset`
- Rule `choose` selects and applies rule if in phase `fire`:
  - `choose/3` selects rule from nonempty `conflictset`
  - Updated `conflictset` without applied rule is added
  - **Then** rule `R` applied by rule `apply/1` (if still possible)
  - **Then** phase constraint `fire` is called again

Conflict resolution strategies (I)

Rule choice depends on the property P of the rules.
Select a rule from the conflict set according to strategy:

- dfs: ensures depth first traversal; fast
- bfs: ensures breadth first traversal; fair
- $random$: selects a rule randomly; fast and fair
- P (a number): selects a rule with highest priority

Conflict resolution strategies (II)

Choice rules

```
dfs @ choose(L,R,L1) ⇔ L=[rule(dfs,_,_)|_] |
   L=[R|L1].

bfs @ choose(L,R,L1) ⇔ L=[rule(bfs,_,_)|_] |
   append(L1,[R],L).

random @ choose(L,R,L1) ⇔ L=[rule(random,_,_)|_] |
   random_select(R,L,L1).

priority @ choose(L,R,L1) ⇔ L=[rule(N,_,_)|_],number(N) |
   sort(L,[R|L1]). % smaller number means higher priority
```

Assumes that all rules have same type of property and strategy

Example - Coin flip

Example (CHR with Random choice)

```
h @ coin <=> head : random.
t @ coin <=> tail : random.
```

Example (CHR Conflict Resolution)

```
find-h @ coin ==> true | conflictset([rule(random,h,[])]).
apply-h @ coin, apply(rule(random,h,[])) <=> head.

find-t @ coin ==> true | conflictset([rule(random,t,[])]).
apply-t @ coin, apply(rule(random,t,[])) <=> tail.
```

Example (CHR Execution)

Query `coin, fire` leads to `conflictset([rule(random,h,[]), rule(random,t,[])])` from which `choose` randomly selects.

Example (Dijkstra's shortest path)

Example (CHR with dynamic Priority)

```
d1 @ dist(X,N) \ dist(X,M) <=> N=<M | true : 0.
dn @ dist(X,N), edge(X,Y,M) ==> N1 is N+M, dist(Y,N1) : N.
```

Example (CHR Conflict Resolution)

```
dist(A,B), dist(A,C) ==> B=<C |
       conflictset([rule(0,d1,[A,B,C])]).
dist(A,B) \ apply(rule(0,d1,[A,B,C])), dist(A,C) <=> true.

dist(A,B), edge(A,C,D) ==>
       conflictset([rule(B,dn,[A,B,C,D])]).
dist(A,B), edge(A,C,D) \ apply(rule(B,dn,[A,B,C,D])) <=>
       F is B+D, dist(C, F).
```

Negation-as-Absence using Conflict resolution

Definition (Rule scheme for CHR rule with negation)

Generalised simpagation rule with negated pattern, property `P` and identifier `Id`

```
H1 \ H2 -(NH,NG) ⇔ C | B : P
```

translates to CHR rules

```
find   @ H1 ∧ H2 ⇒ C | conflictset([rule(P,Id,vars(H1,H2,C))])
ignore @ NH \ apply(rule(P,Id,vars(H1,H2,C))) ⇔ NG | true
apply  @ H1 \ H2 ∧ apply(rule(P,Id,vars(H1,H2,C))) ⇔ B
```

- `find` and `apply` rules as before
- Add `ignore` rule before `apply` rule
  Deletes `apply/1` if `NH` is present and `NG` holds

Example - Marital Status

Example (CHR with Negation))

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

Example (CHR with Conflict Resolution))

```
find @ person(A) ==> conflictset([rule(dfs,st,[A])]).
ignore @ married(A) \ apply(rule(dfs,st,[A])) <=> true.
apply @ person(A) \ apply(rule(dfs,st,[A])) <=> single(A).
```

Summary production rule systems in CHR

- **Conflict Resolution with Negation-as-Absence** transferred to CHR under *refined semantics*
- Conflict Resolution and Negation-as-Absence are **non-monotonic**
- **Propagation rules** compute rule application attempts without side-effects
- **Phase constraints** ensure rules are applied only when present
- Phase constraints rely on *left-to-right evaluation order* of queries
- Phase constraint triggers conflict resolution after each rule application
- Negation-as-Absence relies on *rule order* in program text
- Special case of Negation-as-Absence avoids negation at all
- PGRS-rules subset of CHR expressable in production rule systems

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

Some Database Terminology

- **Attribute:** Argument of a database tuple (relation). Typically scalar data type (constants).
- **Integrity Constraints:** Conditions on attributes in tuples and between tuples that have to hold in a consistent database.
- **Trigger:** An event that is to cause an action.
- **View Maintenance:** Update to some attributes of a tuple should result in meaningful values for the other attributes.
- **Transaction:** an independent sequence of atomic, consistent, isolated and durable database actions. Changes only take effect if transaction ends successfully and is **committed**.
- **Rollback:** Undo transaction and its effects.

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)
- *Conflict resolution* may be necessary
- Concurrent or sequential execution?

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`
- Arguments of these constraints are ground database tuples

Definition (Rule scheme for database relation)

Each n-ary relation r generates CHR rules

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

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

Additional generic rules to remove events at end of program

Definition (Database operation event removal)

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

Embedding ECA rules in CHR with Patterns

Extend semantics by patterns with variables:

- `delete/1` and `update/2` take patterns as arguments
- Update **all** tuples matching pattern (do not bind pattern)

Definition (Rule scheme for database relation)

Each n-ary relation r generates CHR rules

```
ins @ insert(R) ⇒ R
del @ delete(P) \ R ⇔ match(P,R) | true
upd @ update(P,P1) \ R ⇔ match((P,P1),(R,R1)) ∧ R ≠ R1 | R1
```

where `P, P1, R, R1` are of the form $r(x_1, \ldots, x_n)$, all variables distinct and fresh

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

`R ≠ R1` ensures termination of `update/2`

Example (Limit employee's salary increase by 10 %)

```
DEFINE RULE LimitSalaryRaise
IF employee.salary > 1.1 * PREVIOUS employee.salary
THEN update employee.salary = 1.1 * PREVIOUS employee.salary
```

CHR embedding of above SQL statement:

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

Example

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

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

Production Rule Negation Example with Deletion in ECA

Since deletion is explicit in ECA, we can react to it

Example (Marital Status Code)

```
% person(X) -(married(X),true) ==> single(X).
insert(person(A)) ==> insert(single(A)).
married(A) \ insert(single(A)) <=> true.
insert(married(A)), single(A) ==> delete(single(A)).
delete(married(A)), person(A) ==> insert(single(A)).
```

Example (Marital Status Queries)

```
?- insert(person(sue)).
person(sue), single(sue)
?- insert(person(sue)),insert(married(sue)).
person(sue), married(sue)
?- insert(married(sue)),insert(person(sue)),delete(married(sue)).
person(sue), single(sue)
```

Transaction with Commit and Rollback

- Each transaction has a unique identifier.
- Each database update event of a transaction gets this identifier.
- Transactions start with `transaction(Id)` and end with `commit(Id)` or `rollback(Id)`.
- The previous rule scheme is adapted accordingly.

Definition (Database Update Events with Transactions)

Each rule is prefixed by `commit(I)` as phase constraint.

```
ins@ commit(I)∧insert(I,R) ⇒ R
del@ commit(I)∧delete(I,P)\R ⇔ match(P,R) | true
upd@ commit(I)∧update(I,P,P1)\R ⇔ match((P,P1),(R,R1))∧R≠R1 | R1
```

Transaction and clean-up rules come at the end of the program.

```
transaction(I) \ insert(R) ⇔ insert(I,R).
rollback(I) \ insert(I,_) ⇔ true.
commit(I) \ insert(I,_) ⇔ true.
% analogous for delete and update events
```

Localized Constraints, Data and Operation Constraints in CHR

Definition (Localized CHR)

- **Localized constraint:** one argument is an identifier, the **location**.
- **Global constraint:** is a constraint that is not localized.
- **Localized rule:** localized constraints in head have same location and localized constraints in body have same location.

For example, in the CHR embedding, transactions use localized constraints while database relations are global.

Definition (Data and Operation Constraints)

- We may distinguish between **data** and **operation** constraints.
- In the head of each rule, at least one operation constraint.
- The rule (partially) **defines** the operation.

FP functions and database update events in CHR as operation constraints.

Comparison ECA-Rules and CHR-Rules

- ECA-Rules for databases, CHR-Rules for programs
- ECA-Rules with explicit updates, in particular deletion
- Embedding events as operation constraints, database relations as data constraints
- Embedding transactions with localized constraints
- PGRS-Rules of CHR with constants expressable as ECA-Rules

Logical Algorithms 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 for program analysis algorithms
- **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)
- $del(A)$: Permanent deletion 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

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)

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

```
A \ A ⇔ true.                 % Set-Basedness
del(A) \ del(A) ⇔ true.       % Set-Basedness
del(A) \ A ⇔ true.            % Permanent Deletion (by Overshadowing)
```

Definition (Rule scheme for LA rule)

LA rule

$$r \mathbin{@} p : A \to C$$

translates to CHR propagation rules with priority

$$r \mathbin{@} A_1 \Rightarrow A_2 \mid C : p$$

and derived rules from set-based transformation
(A_1: atoms from A, A_2: comparisons from A)

Priorities by conflict resolution (or CHR language extension)

Ensuring set-based semantics by transformation

Generation of new rule variants by unifying head constraints
Applicable to CHR rules in general (written as simplification rules)

Definition (Rule scheme for set-based semantics)

To a CHR generalized simplification rule

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

add each rule (if guard does not imply that head is same as body)

$$H \wedge H_1 \Leftrightarrow H_1 = H_2 \wedge G \mid B'$$

where B' is B with duplicate constraints removed.

Example

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

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

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

LA Set-Basedness by Confluence Analysis

Example

Set Rule

```
a(A,B) \ a(A,B) <=> true.
```

Example Rule

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

Critical Overlap (other overlaps are joinable)

```
a(1,Y), a(X,2), (a(1,Y),a(X,2))=(a(A,B),a(A,B))
```

after simplification

```
a(1,Y), a(X,2), X=1,Y=2
```

Critical Pair

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

after applying set rule to second state

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

For confluence, add rule from first to second state, simplified

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

LA example (Dijkstra's shortest paths)

Example (Dijkstra in LA)

```
d0 @ 0: dist(X,N) ∧ dist(X,M) ∧ N<M → del(dist(X,M))
dn @ N: dist(X,N) ∧ edge(X,Y,M) → dist(Y,N+M)
```

Example (LA 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.
% analogous for edge/3

d0 @ dist(X,N), dist(X,M) ==> N<M | del(dist(X,M)) :0.
dn @ dist(X,N), edge(X,Y,M) ==> dist(Y,N+M) :N.
```

Set-based transformation does not introduce more rules

Permanent deletion of larger distances correct

Production Rule Examples in LA

Set-basedness does not add rules. Permanent deletion problematic.

Example (Minimum in CHR)

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

Number `X` can never be minimum again.

Example (Transitive closure in CHR)

```
% p(X,Y), p(X,Y) ==> del(p(X,Y)) ?
e(X,Y) ==> p(X,Y).
```

Duplicate elimination is already covered by set-basedness.

Example (Marital Status in CHR)

```
married(X), single(X) ==> del(single(X)).
person(X) ==> single(X).
```

Married person can never become single again.

LA Deletion and Confluence

Example (General Schematic Case)

Permanent deletion for some constraint A

```
del(A) \ A ⇔ true.
```

Arbitrary program rule with constraint A

```
A ∧ B ⇒ G | C
```

Overlap of the two rules

```
del(A) ∧ A ∧ B ∧ G
```

Critical pair, simplified

```
del(A) ∧ B ∧ G <> del(A) ∧ B ∧ G ∧ C
```

For confluence, we may add rule deleting all of `C`

```
del(A) ∧ B ⇒ G | del(C)
```

but does not work if `C` contains deletion

Non-Permanent Deletion and Duplicates in LA

Permanent deletion may be problematic. Can we avoid it?

- LA atoms get a unique numeric identifier.
  $\Rightarrow$ LA deletion and Set-Basedness only apply to atoms with same identifier.
  $\Rightarrow$ Same atom can be added back with a new identifier.
- Auxiliary constraint `lastId/1` keeps track of last identifier used. `lastID(0)` is added as first atom to each query.
- In rules, same identifier only for atoms both in head and body. Other identifiers in head are ignored. For the body atoms, new identifiers are generated.

Embedding CHR in LA with Identifiers for Atoms

Definition (Rule scheme for embedding PGRS-CHR in LA)

Generalised simpagation PGRS CHR rule, only comparisons in guard

$$\texttt{H1...Hi \ } \texttt{H}_{i+1}\texttt{...Hn} \Leftrightarrow \texttt{ G | B1 ... Bm}$$

translates to LA rule with numeric identifiers added to LA atoms

$$\begin{aligned}
&\texttt{lastID(Id)} \wedge \\
&\texttt{H1[Id1]...Hi[Idi]} \wedge \texttt{H}_{i+1}\texttt{[Id}_{i+1}\texttt{]...Hn[Idn]} \wedge \texttt{G} \Rightarrow \\
&\qquad \texttt{del(lastID(Id))} \wedge \texttt{lastID(Id+m)} \wedge \\
&\qquad \texttt{del(H}_{i+1}\texttt{[Id}_{i+1}\texttt{])...del(Hn[Idn])} \wedge \\
&\qquad \texttt{B1[Id+1]...Bm[Id+m]}
\end{aligned}$$

where `A[Idj]` stands for `A` with an additional argument `Idj` and where `Id, Id1...Idn` are new different variables.

Deductive Databases

- Deductive database systems can make deductions (inferences) based on stored rules and facts.
- Combination of logic programming and relational databases.
  - More expressive than relational databases (recursion).
  - Less expressive than logic programming systems (constants only).
- Applications: semantic web, data integration, information extraction, program analysis, security, cloud computing, machine learning.

Datalog for Deductive Databases

Datalog is deductive database query language (no updates)

- Logical inference rules similar to Logical Algorithms and Prolog
  - Atoms are **set-based**. Rules are range-restricted
- Only **finite domains** of constants (no function symbols)
- **Nonmonotonic Negation** as in production rule systems and Prolog
- Negation must be **safe**: ground at execution time
- **Stratification**: No recursion through negation
- Datalog is **not Turing complete** (finite data)

Datalog

Datalog Evaluation Strategy

- Evaluated by bottom-up propagation like Logical Algorithms
- For efficiency, evaluation is made query-driven
  - **Magic-Set** transformation adds filter predicates to rules

Influence of Datalog

- Magic Set evaluation implemented in IBM's DB2 system
- Influenced database language SQL to allow for recursive queries

Embedding Datalog in CHR

Definition (Rule scheme for Datalog predicate)

Each n-ary Datalog predicate a/n generates simpagation rule
($A = a(x_1, \ldots, x_n)$ with x_i distinct variables)

```
A \ A ⇔ true.     % Set-Basedness
```

Definition (Rule scheme for Datalog rule and fact)

Datalog rule $C :- A$ or **fact (tuple)** C translate to propagation rule

$A_1 \Rightarrow A_2 \mid C$ (if A_1 is empty, use auxiliary constraint db for A_1)

and derived rules from set-based transformation (see LA)

(A_1: atoms from A, A_2: built-ins from A, C: single user atom)

Analogous to translation of Logical Algorithms to CHR

Stratified safe **negation** by negation-as-absence from PR

Examples with Negation in Datalog

Example (Shaving paradox in Datalog)

Non-stratified but safe negation (not allowed in Datalog)

```
shaves(barber,X) :- man(X), -(shaves(X,X),true).
```

Consider query `man(barber)` (Russell's paradox)

Example (Minimum in Datalog)

Stratified but un-safe negation (not allowed in Datalog)

```
min(X) :- num(X), -(num(Y),Y<X).
```

Example (Marital status in Datalog)

Stratified and safe negation, translated as negation-as-absence

```
% single(X) :- person(X), -(married(X),true).
fire, person(X) ==> check(status,[X]).
married(X) \ check(status,[X]) <=> true.
person(X)  \ check(status,[X]) <=> single(X).
```

Example Datalog in CHR

Example (Transitive Closure in Datalog)

```
p(X,Y) :- e(X,Y).
p(X,Z) :- e(X,Y), p(Y,Z).
```

Example (Transitive Closure in Datalog-CHR)

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

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

Set-based rule transformation does not introduce more rules

Datalog Magic Set Transformation Optimization (I)

Generate only tuples (constraints) relevant to query demand pattern `q(A)`

In pattern `A`, variables stand for arbitrary values and will not be bound

Definition (Magic Set Query Generation)

Add query pattern subsumption rule (first in the program for termination)

```
subsume @ q(A) \ q(B) <=> match(A,B) | true
```

For each Datalog PGRS CHR propagation rule (`B` single CHR constraint)

```
H1...Hn ⇒ G | B
```

add query pattern generating inverted backwards rule

```
q(A) ⇒ copy(A,B),G | q(H1)...q(Hn)
```

First rule keeps more general pattern, `match(A,B)` holds if `B` instance of `A`

Built-in `copy(A,B)` unifies `B` with copy of `A`, to intersect patterns `A` and `B`

Datalog Magic Set Transformation Optimization (II)

Filter database tuples (constraints) according to query demand pattern `q(A)`.

Definition (Magic Set Tuple Filter)

Each Datalog PGRS CHR propagation rule (`B` single CHR constraint)

```
H1...Hn ⇒ G | B
```

is replaced by query-filtered rule

```
H1...Hn, q(A) ⇒ copy(A,B),G | B
```

Datalog propagation rule now fires only if needed, if query pattern demands it

Example Datalog Magic Set Transformation Optimization

Example (Transitive Closure)

Given Datalog CHR propagation rules

```
e(X,Y) ==> p(X,Y).
e(X,Y), p(Y,Z) ==> p(X,Z).
db ==> e(1,2).   db ==> ...   ...
```

are replaced by the propagation rules

```
        % query generation rules (for db/0 not needed)
q(A) ==> copy(A,p(X,Y)) | q(e(X,Y)).
q(A) ==> copy(A,p(X,Z)) | q(e(X,Y)), q(p(Y,Z)).
        % query-filtered rules
e(X,Y), q(A) ==> match(A,p(X,Y)) | p(X,Y).
e(X,Y), p(Y,Z), q(A) ==> match(A,p(X,Z)) | p(X,Z).
db, q(A) ==> match(A,e(1,2)) | e(1,2).  db, q(A) ==> ...   ...
        % sample query  ?- db, q(p(_,2))
```

Example Queries Datalog Magic Set

Example (Transitive Closure)

Given the Datalog database facts (tuples)

```
db ==> e(1,2).  db ==> e(2,3).  db ==> e(3,4).  db ==> e(4,2).
```

Some queries and their results

```
?- db, q(p(4,1)).
% no path found
q(e(_,_)), q(p(_,1)), e(1,2),e(2,3),e(3,4),e(4,2)
% actually all edges and no path ending in 1 found

?- db, q(p(4,3)).
% path found
q(e(_,_)), q(p(_,3)), e(1,2),e(2,3),e(3,4),e(4,2),
p(1,3), p(2,3), p(3,3), p(4,3)
% all paths ending in 3 needed to compute result
```

Datalog Magic Set Transformation Optimization (III)

Definition (Magic Set for Negation)

Each Datalog PGRS CHR propagation rule with negation (`N` unique identifier)

`N @ H1...Hn, -(NH,NC) ⇒ G | B` (`B` single CHR constraint)

is replaced by query pattern generating inverted backwards rule

`q(A) ⇒ copy(A,B) | q(H1)...q(Hn), q(NH)`

and by rules for negation-as-absence with `q/1` (only rule `N1` query-filtered)

```
N1 @ fire, H1...Hn, q(A) ⇒ G,match(A,B) | check(N,Vars)
N2 @ NH \ check(N,Vars) ⇔ NC | true
N3 @ H1...Hn \ check(N,Vars) ⇔ B
```

where `Vars` are the variables from the lhs `H1...Hn,G`.

Example Datalog Magic Set with Negation

Example (Marital Status)

Given Datalog CHR propagation rules

```
person(X), -(married(X),true) ==> single(X).
db ==> person(sue).  db ==> ...   ...
```

are replaced by the propagation rules

```
        % query generation rules (for db/0 not needed)
q(A) ==> copy(A,single(X)) | q(person(X)), q(married(X)).
        % query-filtered rules with negation-as-absence
fire, person(X), q(A) ==> match(A,single(X)) | check(status,[X]).
married(X) \ check(status,[X]) <=> true.
person(X)  \ check(status,[X]) <=> single(X).
db, q(A) ==> match(A,person(sue)) | person(sue).
db, q(A) ==> ...   ...
        % sample query  ?- db, q(single(_)), fire
```

Example Queries Datalog Magic Set with Negation

Example (Marital Status)

Given the Datalog database facts (tuples)

```
db ==> person(sue).     db ==> person(sam).
db ==> married(sam).    db ==> married(fred).
```

Some queries and their results

```
?- db, q(married(sue)), fire.
db, q(married(sue))    % no fact found

?- db, q(married(_)), fire.
db, q(married(_)), married(fred), married(sam)

?- db, q(single(_)), fire.
db, q(married(_)), q(person(_)), q(single(_)),
person(sam),person(sue), married(fred),married(sam), single(sue)
```

Constraint-based and logic-based programming

- Rule-based programming languages
  - Rules with logical variables subject to built-ins (like CHR)
  - Rules without multiple head and body atoms
  - Developed since the 1980's
- Constraint Logic Programming (CLP), Prolog
  - Combines declarativity of logic programming and efficiency of constraint solving
  - Prolog is CLP with syntactic equality as only built-in constraint
  - with negation-as-failure and disjunction for search
- Concurrent Constraint Programming (CCP)
  - Ask-and-Tell metaphor of communication by constraints
  - Approach closest to CHR
  - Guards like CHR

Prolog and Constraint Logic Programming

- Top-down evaluation (Datalog, LA have bottom-up evaluation)
- Don't-know nondeterminism by choice of rule (or disjunct)
- Nonmonotonic Negation-as-failure (like negation-as-absence)

Definition (CLP program)

CLP program: multi-set of Horn clauses

$$A \leftarrow G$$

where A is single atom and G is conjunction of atoms, built-ins and negated conjunctions of the form $neg(G)$)
Fact A is written as clause $A \leftarrow true$

CLP in CHR with disjunction ($CHR^{\vee}$)

- **$CHR^{\vee}$**: CHR language extension with disjunction in body
- 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

Declarative semantics of CLP program as logic formula according to closed-world assumption (CWA)

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)

Clark's completion for Negation

Logically negate formula of Clark's completion

Definition (Rule scheme for negation with Clark's completion)

Negation 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

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

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

CLP embedding in CHR (I)

Direct implementation of declarative semantics

Definition (Rule scheme for CLP clauses with negation)

For each predicate p/n add Clark's completion of p/n as rule with disjunction

`p(`$\bar{x}$`) <=> t1=`$\bar{x}$`,G1 ; ... ; tm=`$\bar{x}$`,Gm`

and add negation of Clark's completion of p/n as two rules for `neg/1`

`neg(p(`$\bar{x}$`)) <=> neg((t1=`$\bar{x}$`,G1)),...,neg((tm=`$\bar{x}$`,Gm))`

`neg((p(`$\bar{x}$`),G)) <=> neg((t1=`$\bar{x}$`,G1,G)),...,neg((tm=`$\bar{x}$`,Gm,G))`

Definition (Generic rules for negation of deterministic built-ins)

Add generic rules in given order. Built-ins deterministic: at most one answer.

`neg(GH) <=>` **copy** `(GH,(G,H)),builtin(G),call(G) | neg(H)`

`neg(G) <=> builtin(G),call(G) | false`

`neg(G) <=> true %` (may be nonmonotonic if `G` not ground)

(`G,H,GH` are variables, `builtin(G)` holds if `G` builtin, `copy/2` as before)

Example CLP embedding in CHR

Example (Append in Prolog)

```
append([],L,L).
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).

neg(append(X,Y,Z)) <=> neg((X=[],Y=L,Z=L)),
   neg((X=[H|L1],Y=L2,Z=[H|L3],append(L1,L2,L3))).
neg((append(X,Y,Z),G)) <=> neg((X=[],Y=L,Z=L,G)),
   neg((X=[H|L1],Y=L2,Z=[H|L3],append(L1,L2,L3),G)).
```

CLP embedding in CHR (II)

Direct consequence of declarative semantics: if for a query for p/n the built-ins for all clauses except one clause fail, use this one clause.

Definition (Rule scheme for deterministic cases)

For each predicate p/n use Clark's completion of p/n to add `m` rules, one for each original clause `i` of p/n

```
det_i @ p(x̄) <=>
  \+((t1=x̄,C1)),...,\+((tm=x̄,Cm)) | % without \+((ti=x̄,Ci))
  ti=x̄,Gi
```

where `\+` is the built-in negation of CLP, the `Ci` are the built-ins of `Gi`

Rules are added as short-cuts before other rules of translation.

Example deterministic CLP embedding in CHR

Example (Append in Prolog)

```
append([],L,L).
append([H|L1],L2,[H|L3]) :- append(L1,L2,L3).
```

Example (Deterministic Append in CHR)

```
append(X,Y,Z) <=> \+((X=[H|L1], Y=L2, Z=[H|L3])) |
                  X=[], Y=L, Z=L.
append(X,Y,Z) <=> \+((X=[], Y=L, Z=L)) |
                  X=[H|L1], Y=L2, Z=[H|L3], append(L1,L2,L3).
```

Sample Queries with deterministic rules alone

```
?- append([1],[2],L).
   L = [1,2]
?- append(L1,L2,[1,2]).
   append(L1,L2,[1,2])
?- append([a],_,[b]).
   false
```

Magic Set Transformation for CLP in CHR

Use variant of Datalog Magic Set transformation with negation

$\Rightarrow$ computes all answers bottom-up at once, keeps intermediate results

Definition (Magic Set Transformation for CLP clauses)

Replace each CLP clause (`G` deterministic built-ins, `A,B1..Bn` atoms)

```
A :- G, B1..Bn
```

by query pattern generating rule (if `n>0`)

```
q(A1) ⇒ copy(A1,A),G | q(B1)..q(Bn)
```

and by query-filtered rule in the other direction

```
B1..Bn, q(A1) ⇒ copy(A1,A),G | A
```

Built-in *copy/2* is used to intersect patterns

Example CLP Magic Set Transformation

Example (CLP Append in CHR)

Prolog clauses

```
ap([],L,L).
ap([H|L1],L2,[H|L3]) :- ap(L1,L2,L3).
```

are replaced by CHR propagation rules

```
        % query generation rules
q(A) ==> copy(A,ap([B|C],D,[B|E])) | q(ap(C,D,E)).

        % query-filtered rules
q(A) ==> copy(A,ap([],B,B)) | ap([],B,B).
ap(C,D,E),q(A) ==> copy(A,ap([B|C],D,[B|E])) | ap([B|C],D,[B|E]).
```

Example Queries CLP Magic Set

Example (CLP Append in CHR)

```
?- q(ap(a,_,_)).
q(ap(a,_,_))  % no answer produced, i.e. failure

?- q(ap([1],[2],L)).
q(ap([1],[2],_)), q(ap([],[2],_))
ap([1],[2],[1,2]), ap([],[2],[2])

?- q(ap(L1,L2,[1,Y])).
q(ap(_,_,[1,Y])), q(ap(_,_,[Y])), q(ap(_,_,[]))
ap([],[1,Y1],[1,Y1]), ap([1],[Y2],[1,Y2]), ap([1,Y3],[],[1,Y3])
ap([],[Y2],[Y2]), ap([Y3],[],[Y3]), ap([],[],[])

?- q(ap(L1,L2,L3)).  % nontermination
```

If there are infinitely many answers, magic set computation does not terminate

Example CLP Magic Set Transformation

Example (CLP Minimum in CHR)

```
% Given Prolog clauses
% m(X,Y,X) :- X=<Y.
% m(X,Y,Y) :- X>=Y.
% no query generation rules
% query-filtered rules
q(A) ==> copy(A,m(X,Y,X)),X=<Y | m(X,Y,X).
q(A) ==> copy(A,m(X,Y,Y)),X>=Y | m(X,Y,Y).
```

Sample queries

```
?- q(m(1,2,_)).
   q(m(1,2,_)), m(1,2,1)
?- q(m(1,1,_)).
   q(m(1,1,_)), m(1,1,1), m(1,1,1) % two identical answers
?- q(m(1,0,1)).
   q(m(1,0,1))  % no answer produced, i.e. failure
?- q(m(1,_,1)). % not sufficently instantiated
```

Textbook Example (Prime sieve)

Comparison between Prolog and CHR by example

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 nonmonotonic 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.
```

Textbook Example (Shortest path)

Comparison between Prolog and CHR by example

Example (Shortest path in Prolog)

```
p(From,To,Path,1) :- e(From,To).
p(From,To,Path,N) :- e(From,Via),
                     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 nonmonotonic 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 dont-care and dont-know nondeterminism
  - Consider the committed-choice search-free fragment of CC (don't-care nondeterminism like CHR)

Definition (Abstract syntax of CC program)

CC program is a finite sequence of declarations

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

(p user-defined predicate symbol, $\bar{t}$ 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)
  - Checks whether constraint is implied by constraint store
  - Realized by simple test if constraint ground

Generalizes idea of concurrent data flow computations

- Operation (constraint) 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 store d and continue with associated 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 $\|$ 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 $p(\bar{t})$ by its definition

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

Embedding in CHR

Translation scheme maps

- 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/1`
- CC Ask constraint $\rightarrow$ built-in in guard of CHR rule
- CC Tell constraint $\rightarrow$ built-in in body of CHR rule

Translation

Definition (Rule scheme for CC expressions)

CC declarations and agents

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

embedded in CHR with single-headed simplification rules

$$\begin{array}{lll} \text{Program} & D ::= & p(\bar{x}) \Leftrightarrow \bar{x}{=}\bar{t} \wedge A \mid D, D \ (\bar{x} \text{ fresh distinct variables}) \\ \text{Goals} & A ::= & \mathit{true} \mid c \mid \mathtt{ask}(\sum_{i=1}^{n} c_i \rightarrow A_i) \mid A \wedge A \mid p(\bar{t}) \end{array}$$

For each Ask goal constraint $\mathtt{ask}(\sum_{i=1}^{n} c_i \rightarrow A_i)$ generate n single-headed simplification rules

$$\mathtt{ask}(\sum_{i=1}^{n} c_i \rightarrow A_i) \Leftrightarrow c_i \mid A_i \quad (1 \leq i \leq n)$$

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.
```

Simplify ask rules: introduce unique constraint symbol for each ask constraint goal, keep only variables. Program transformation replaces `ask((X≤Y→Y=Z)+(Y≤X→X=Z))` by `ask_max(X,Y,Z)`

Example (Simplified maximum in CHR)

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

Embedding rule-based approaches in CHR

Using source-to-source transformation for embeddings

- Rewriting- and graph-based **formalisms**
  - GAMMA Multiset Transformation
  - Term Rewriting Systems and Functional Programming (also a language)
  - Colored Petri Nets
- Rule-based **systems**
  - Production Rules
  - Event-Condition-Action Rules
  - Logical Algorithms (also a formalism)
- Logic- and constraint-based **programming languages**
  - Datalog for Deductive Databases (also a system)
  - Prolog and Constraint Logic Programming
  - Concurrent Constraint Programming

Advantages of Embeddings in CHR

Advantages of CHR for **execution**

- Effective and efficient high-level embeddings
- Abstract (non-ground) execution for compile-time partial evaluation and program transformation
- Incremental, anytime, online algorithms for free
- Concurrent, parallel for confluent programs
- Completion can make programs confluent

Advantages of CHR for **analysis**

- Decidable confluence and operational equivalence
- Estimating complexity semi-automatically
- Logic-based declarative semantics for correctness

Embeddings for comparison and cross-fertilization (transfer of ideas)

Positive ground range-restricted CHR

- Approaches can be embedded into simple CHR fragment (except constraint logic programming and concurrent programming)
  - **positive**: no built-ins in body of rule
  - **ground**: queries ground
  - **range-restricted**: variables in body also occur in head
    or as result of auxiliary functions as non-failing built-ins in body
- These conditions imply
  - Every state in a computation is ground
  - CHR constraints need to do not wake up
  - Guard entailment check is just test
  - Computations cannot fail

Additional Features of Embeddings

CHR implementations and language extensions exist for

- **non-monotonic negation** of rule-based systems and constraint logic programming
- **conflict resolution** of rule-based systems (rule priorities)
- **explicit deletion** of rule-based systems
- **set-based semantics** of Logical Algorithms and Datalog
- **bottom-up and top-down evaluation** combined by magic sets of Datalog and constraint logic programming
- **built-in search** of constraint logic programming
- **diagrammatic notation** of graph-based systems (Petri Nets)

Features of CHR for programming

Unique combination of features in CHR

- **Multiple head and body atoms** only in rule-based formalisms and rule-based systems
  - Avoid use of loops and recursion for iteration
- **Propagation rules** only in rule-based systems, constraint logic programming languages
  - Make implicit information explicit
- **Non-ground constraints** only in constraint-based languages
  - Constraint solving with logical variables
  - Notion of failure due to inconsistent built-in constraints
- **Logical Declarative Semantics** only in term rewriting systems, constraint logic programming
  - Computations justified by logic reading of rules as formulas

Embedding fragments of CHR in other rule-based approaches

- **Positive ground range-restricted fragments** embeddable into
  - *Rule-based systems*
  - **Only simplification rules** in *rule-based formalisms*
    - GAMMA
    - Only tree-structured simplification rules in term rewriting
    - Only single-headed simplification rules in functional programming
  - **Only rules over finite data** (not Turing-complete) in Datalog and Petri Nets
- **Non-ground fragments** embeddable into
  - **Single-headed simplification rules** in
    - *Constraint-based programming languages*
  - **Single-bodied propagation rules** in
    - Constraint logic programming languages

The Potential of Constraint Handling Rules

Rule-based systems, formalisms and languages can be compared and cross-fertilize each other via CHR

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

CHR - an essential unifying computational formalism

CHR - a Lingua Franca for computer science