Constraint Handling Rules - Program analysis

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

Table of Contents

Overview (I)

- Advantage of declarative languages: Ease of formally sound program analysis
- Confluence and program equivalence decidable for terminating CHR programs
- Following results apply to (very) abstract semantics
  - Not all results carry over to refined semantics

Overview (II)

- Termination in CHR is undecidable (Turing completeness)
- Confluence
  - Nondeterminism does not matter for result
  - Relation between initial and final state is function
  - Test for confluence for terminating programs
  - Logical correctness is implied by confluence
  - Soundness and completeness improved
  - Parallelization without change of program
- Program equivalence
  - Decidable test for operational equivalence
  - No other practical language known with such a test

Termination

Each rule head is larger than rule body in well-founded termination order

Definition (Well-foundedness)

- Order $\gg$ is well-founded if no infinite descending chains $t_1 \gg t_2 \gg \ldots \gg \ldots$ exist
- (CHR) termination order is well-founded order on CHR states

Termination for simplification rules

- Relation $>$ on natural numbers used as basis of termination order
- Rankings
  - Mapping of logical expressions to natural numbers
  - Give also upper bounds for worst-case derivation lengths

Rankings

- Arithmetic function mapping terms and formulas to integers
  - Inductively defined on function, constraint symbols and logical connectives
- Resulting order on formulas is total
  - Order is well-founded if smallest value exists
- Linear polynomials as ranking functions
  - Rank of expression defined by linear positive combination of argument ranks

Definition

Definition (Ranking)

Formula B: $rank(B) \geq 0$ required

Built-in C: $rank(C) = 0$ required

Conjunction: $rank((A \wedge B)) = rank(A) + rank(B)$

Function/constraint symbol f with arguments t_i ($a_i^f \in \mathbb{N}$):

$$rank(f(t_1, \ldots, t_n)) = a_0^f + a_1^f * rank(t_1) + \ldots + a_n^f * rank(t_n)$$

$rank(s) > rank(t)$ is order constraint

Syntactic size

Definition (Syntactic size)

Syntactic size of term can be expressed as the ranking

$$size(f(t_1, \ldots, t_n)) = 1 + size(t_1) + \ldots + size(t_n)$$

Example

- $size(\texttt{f(a,g(b,c))}) = 5$
- $size(\texttt{f(a,X)}) = 2 + size(\texttt{X})$ where $size(\texttt{X}) \geq 0$
- $size(\texttt{f(g(X),X)}) \geq size(\texttt{a,X})$ because $2 + 2 * size(\texttt{X}) \geq 2 + size(\texttt{X})$

Rankings of constraints and conjunctions

- Rankings of CHR and built-in constraints
  - Built-in has rank 0 (termination established)
  - Built-in may imply order constraints on arguments (e.g. $s = t \rightarrow rank(s) = rank(t)$)
- Rankings of conjunctions using conjuncts:
  - Sum properly reflects worst case derivation length
  - Takes properties of conjunction into account (associativity, commutativity, not impotence)

Ranking of simplification rule

Definition (Ranking of simplification rule)

Ranking (condition) of simplification rule $H \Leftrightarrow G \mid B$ is

$$\forall\ (C \rightarrow rank(H) > rank(B))$$

(C conjunction of rule's built-ins, $rank$ CHR ranking function)

- Idea: built-ins imply order constraints helping to show $rank(H) > rank(B)$
- Built-ins in guard do not hold: rule not applicable
- Built-ins in body do not hold: inconsistent, thus final state

Example

Example

`even(s(N)) ⇔ N=s(M)∧even(M).`

- `even(N)` and `even(0)` delay
- `even(s(N))` results in `N=s(M), even(M)`
- `even(s(0))` fails since `0=s(M)`

Suitable polynomial interpretation

$$rank(even(N)) = size(N)$$

Resulting ranking condition (holds) for rule

$$N = s(M) \rightarrow rank(s(N)) > rank(M)$$

Program termination

Definition (Bounded goal)

Goal G **bounded** if rank of any instance of G bounded above by constant

- Rank of ground term always bounded
- In bounded goals variable appear only in positions which are ignored by ranking
- Not bounded query might lead to nontermination (e.g. `even(n),even(s(N))`)

Bounded goals and termination

Idea: Rank of removed constraints is higher than rank of added constraints

Theorem

If P consist only of simplification rules and ranking condition holds for each rule in program P, then P is terminating for every bounded goal

Suitable ranking with suitable order constraints cannot be found automatically (termination undecidable)

Derivation lengths

- Rank of query gives upper bound on number of rule applications (derivation length)
- In previous example: argument of `even` decreases by 2 with every rule application

Theorem

If ranking condition holds for each rule in P, containing only simplification rules, then the worst-case derivation length D for bounded goal G in P is bounded by rank of G:

$$D(G) \leq rank(G)$$

Confluence

- Confluence guaranties that any computation for a goal results in same final state
  - Independent from order of rule applications
  - Independent from order of rules in program and constraints in goal
- Decidable, sufficient, and necessary test for confluence in CHR
  - Returns conflicting rule applications

Minimal states (I)

- Each rule has most general state it is applicable to
- Removing any constraint from this minimal state would make rule inapplicable
- Rule still applicable when adding constraints (monotonicity)

Definition (Minimal state)

Minimal state of a rule is the conjunction of its head and guard

- Decidable program analysis: consider minimal states instead of infinitely many

Minimal states (II)

Theorem (Containing minimal states)

If $H'_1 \wedge H'_2 \wedge C$ is minimal state of rule r and r is applicable to state S then there exists goal G' such that $H'_1 \wedge H'_2 \wedge C \wedge G' \equiv S$ (S contains minimal state)

- All states to which rule is applicable contain minimal state
- Logical reading of these states imply logical reading of minimal state

Joinability

- Defines what it means for two derivations to lead to the same result
- Applies to every transition system

Definition (Joinability)

States S_1, S_2 joinable if there are states S_1', S_2' such that $S_1 \mapsto^* S_1'$ and $S_2 \mapsto^* S_2'$ and $S_1' \equiv S_2'$

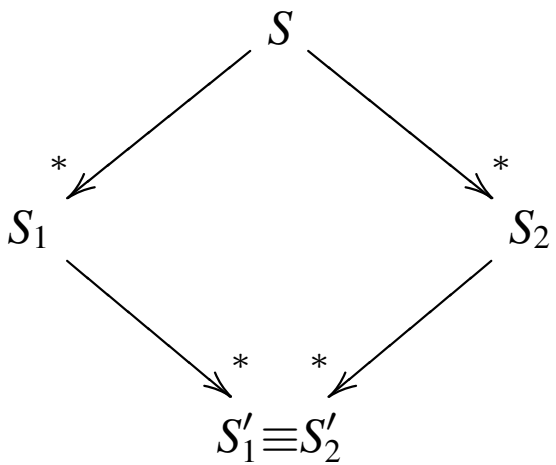
Confluence diagram

Definition (Confluence, well-behavedness)

- CHR program is **confluent** if for all state S, S_1, S_2

  If $S \mapsto^* S_1, S \mapsto^* S_2$ then S_1 and S_2 are joinable.

- CHR program is **well-behaved** if terminating and confluent



Confluence test

- Analyzing joinability of infinitely many states impossible
- But: analysis of only finite number of most general states (overlaps) required for terminating programs
- Overlaps
  - States where more than one rule is applicable
  - Consists of minimal states (heads and guards) of rules
  - Can be extended to arbitrary states by adding constraints

Overlaps

Definition (Overlap)

R_1 simplification or simpagation rule, R_2 rule (renamed apart), $H_i \wedge A_i$ conjunction of head constraints, C_i guard, B_i body.
A (nontrivial) overlap S of rule R_1 and R_2 is

$$S = (H_1 \wedge A_1 \wedge H_2 \wedge (A_1{=}A_2) \wedge C_1 \wedge C_2)$$

if A_1 and A_2 nonempty and $\mathcal{CT} \models \exists((A_1{=}A_2) \wedge C_1 \wedge C_2)$

Critical pairs

Definition (Critical pair)

$S_1 = (B_1 \wedge H_2 \wedge (A_1 = A_2) \wedge C_1 \wedge C_2)$ and $S_2 = (H_1 \wedge B_2 \wedge (A_1 = A_2) \wedge C_1 \wedge C_2)$
Then the tuple (S_1, S_2) is a critical pair (c.p.) of R_1 and R_2.
Critical pair (S_1, S_2) is joinable, if S_1 and S_2 joinable.

- Critical pair results from applying rule to overlap
  - $S \mapsto S_1$ using R_1
  - $S \mapsto S_2$ using R_2

Joinability

- Joinability of critical pairs necessary condition for confluence
- Joinability destroyed only if one rule inhibits application of others (removing head constraints matched by other)
- Critical pairs of propagation rules always joinable
- To destroy joinability one rule must not be propagation rule and rules must overlap
- Nonjoinable critical pair is counterexample for confluence

Joinability for confluence (I)

Definition (Local confluence)

CHR program locally confluent if for all state S, S_1, S_2

$$\text{If } S \mapsto S_1, S \mapsto S_2 \text{ then } S_1 \text{ and } S_2 \text{ are joinable}$$

Theorem (Newman's Lemma)

Terminating reduction system is confluent iff it is locally confluent

Theorem (Confluence)

Terminating CHR program is confluent iff all its critical pairs are joinable

Joinability for confluence (II)

- Theorem gives decidable, sufficient, and necessary condition for confluence
- Joinability of critical pairs not only necessary but also sufficient condition for confluence of terminating programs
- Joinability of c.p. is decidable because program terminating and only finitely many c.p.

Examples(I)

Example

$\mathtt{p} \Leftrightarrow \mathtt{q}.$

$\mathtt{p} \Leftrightarrow \mathit{false}.$

- One overlap $\mathtt{p}$
- Critical pair $(\mathtt{q},\ \mathit{false})$: final and different, thus nonjoinable final states

Example (Coin throw)

$\mathtt{throw(Coin)} \Leftrightarrow \mathtt{Coin} = \mathtt{head}.$

$\mathtt{throw(Coin)} \Leftrightarrow \mathtt{Coin} = \mathtt{tail}.$

- One overlap (after simplifying) $\mathtt{throw(Coin)}$
- Critical pair $(\mathtt{Coin{=}head}, \mathtt{Coin{=}tail})$: nonjoinable states

Examples (II)

Example

`p(X)` $\wedge$ `q(Y)` $\Leftrightarrow$ *true*.

- Overlap of rule with itself: `p(X)` $\wedge$ `q(Y1)` $\wedge$ `q(Y2)`
- Critical pair (`q(Y1), q(Y2)`), `Y1` and `Y2` different variables from overlap
- Analogous situation for overlap `p(X1)` $\wedge$ `p(X2)` $\wedge$ `q(Y)`
- Nonjoinability does not arise for rule

  `p(X)` $\wedge$ `q(Y)` $\Leftrightarrow$ `X=Y` | *true*.

Examples(III)

Example (Destructive assignment)

```
assign(Var,New), cell(Var,Old) <=> cell(Var,New).
```

- Nonjoinable overlap `assign(Var,New1), assign(Var,New2), cell(Var,Old)`
- Results in either `cell(Var,New1)` or `cell(Var,New2)`

Examples (IV)

Example (Maximum)

$\texttt{max(X,Y,Z)} \Leftrightarrow \texttt{X} \leq \texttt{Y} \mid \texttt{Y} = \texttt{Z}.$

$\texttt{max(X,Y,Z)} \Leftrightarrow \texttt{Y} \leq \texttt{X} \mid \texttt{X} = \texttt{Z}.$

- Only overlap: $\texttt{max}(\texttt{X},\texttt{Y},\texttt{Z}) \wedge X \leq Y \wedge Y \leq X$
- Critical pair: $(Y{=}Z \wedge X \leq Y \wedge Y \leq X, X{=}Z \wedge X \leq Y \wedge Y \leq X)$
- Joinable: both states equivalent to $X{=}Y \wedge Y{=}Z$

Examples (V)

Example (Merge (I))

`merge([],L2,L3)` $\Leftrightarrow$ `L2=L3.`

`merge(L1,[],L3)` $\Leftrightarrow$ `L1=L3.`

- Eight critical pairs, some joinable, some not
- Critical pair from first two rules:

  `([]=L1 ∧ L2=[] ∧ L2=L3, []=L1 ∧ L2=[] ∧ L1=L3)`

  - Joinable: Both states equivalent to `L1=[]∧L2=[]∧L3=[]`

Examples (V)

Example (Merge (II))

`merge([X|R1],L2,L3)` $\Leftrightarrow$ `L3=[X|R3]` $\wedge$ `merge(R1,L2,R3).`

`merge(L1,[Y|R2],L3)` $\Leftrightarrow$ `L3=[Y|R3]` $\wedge$ `merge(L1,R2,R3).`

- Critical pair from third and forth rule:

  `(L1=[X|R1]` $\wedge$ `L2=[Y|R2]` $\wedge$ `L3=[X|R3]` $\wedge$ `merge(R1,L2,R3),`

  `L1=[X|R1]` $\wedge$ `L2=[Y|R2]` $\wedge$ `L3=[Y|R3]` $\wedge$ `merge(L1,R2,R3))`

- E.g. query `merge([a],[b],L)` can result in two lists `L`

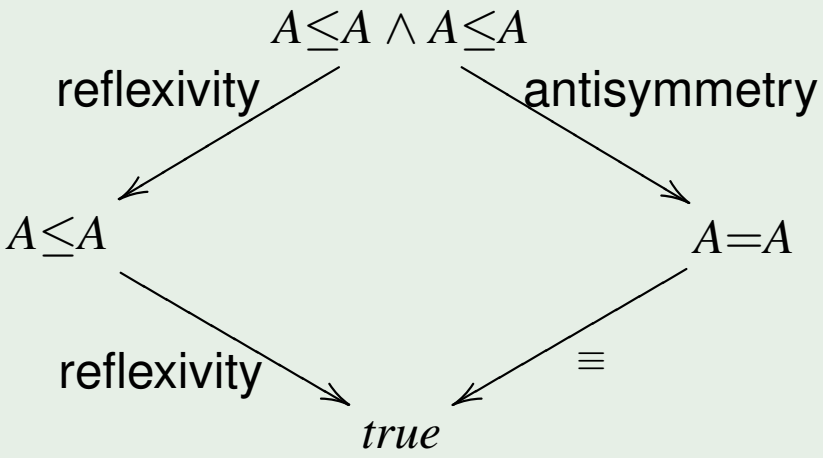
$\Rightarrow$ not confluent (order of elements in list `L` not determined)

Examples (VI)

Example (Partial order constraint (I))

```
reflexivity @ X≤X ⇔ true.
antisymmetry @ X≤Y ∧ Y≤X ⇔ X=Y.
```

- Overlap of reflexivity and antisymmetry



- Critical pair joinable (multiplicity matters in CHR)

Examples (VII)

Example (Partial order constraint (II))

duplicate @ X≤Y ∧ X≤Y ⇔ X≤Y.

antisymmetry @ X≤Y ∧ Y≤X ⇔ X=Y.

transitivity @ X≤Y ∧ Y≤Z ⇒ X≤Z.

- Overlap of antisymmetry and transitivity (left-most head constraint): $\mathtt{X}\leq\mathtt{Y} \wedge \mathtt{Y}\leq\mathtt{Z} \wedge \mathtt{Y}\leq\mathtt{X}$
- Critical pair: $(\mathtt{X}\leq\mathtt{Y} \wedge \mathtt{Y}\leq\mathtt{X} \wedge \mathtt{Y}\leq\mathtt{Z} \wedge \mathtt{X}\leq\mathtt{Z}, \mathtt{X}=\mathtt{Y} \wedge \mathtt{X}\leq\mathtt{Z})$
- Joinable (first state leads to second state):

$$\underline{\mathtt{X}\leq\mathtt{Y}} \wedge \underline{\mathtt{Y}\leq\mathtt{X}} \wedge \mathtt{Y}\leq\mathtt{Z} \wedge \mathtt{X}\leq\mathtt{Z} \quad \mapsto_{\textit{apply antisymmetry}}$$

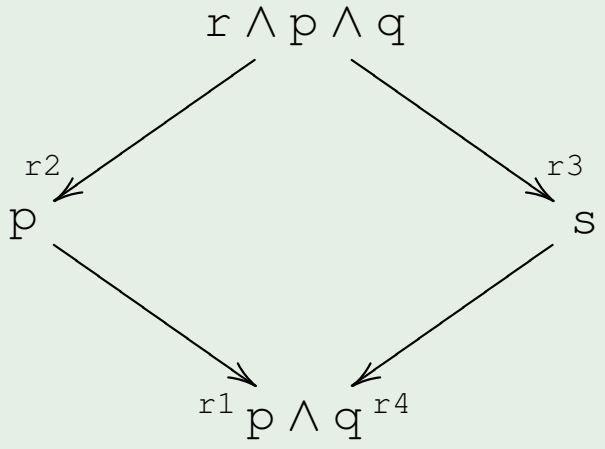
$$\underline{\mathtt{Y}\leq\mathtt{Z}} \wedge \underline{\mathtt{X}\leq\mathtt{Z}} \wedge \mathtt{X}=\mathtt{Y} \quad \mapsto_{\textit{apply duplicate}}$$

$$\mathtt{X}\leq\mathtt{Z} \wedge \mathtt{X}=\mathtt{Y}$$

Confluence test for abstract semantics (I)

Example (Joinable states)

r1 @ p ⇒ q.
r2 @ r∧q ⇔ *true*.
r3 @ r∧p∧q ⇔ s.
r4 @ s ⇔ p∧q.

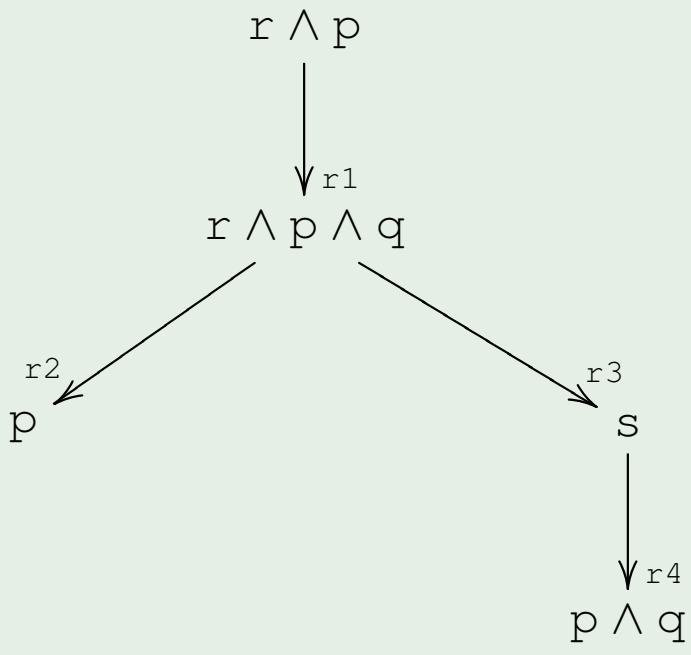


Computation leads to final state p ∧ q no matter which rule applied

Confluence test for abstract semantics (II)

Example (Nonjoinable states)

r1 @ p ⇒ q.
r2 @ r∧q ⇔ *true*.
r3 @ r∧p∧q ⇔ s.
r4 @ s ⇔ p∧q.



- No reapplication of r1 possible to remove p in left branch
- r1 can be applied to p ∧ q (but q cannot be removed)

Confluence test for abstract semantics (III)

- Consider propagation history for abstract semantics
- Propagation history is chosen in states of c.p. such that
  - Application of propagation rules only involving already present constraints not allowed
  - Motived by minimality of states
  - Covers case where all propagation rules already applied before overlap is reached

Confluence test for abstract semantics (IV)

- Associate overlap $S \wedge B$ with ω_t state $\langle \emptyset, S', B, prop(S') \rangle_n^{\mathcal{V}}$
  - S': numbered CHR constraints such that $S = chr(S')$,
  - $\mathcal{V}$: all variables of overlap
  - $prop(S')$ returns propagation history containing entry for each propagation rule with each valid combination of constraints in S'

Detecting nonconfluence

Example (Detecting nonconfluence)

- Overlap `r ∧ p ∧ q`
- Associated ω_t state $\langle \emptyset, \{r\#1, p\#2, q\#3\}, true, \emptyset \rangle_4^{\emptyset}$
- Resulting c.p. $(\langle \{true\}, \{p\#2\}, true, T \} \rangle_4^{\emptyset}, \langle \{s\}, \emptyset, true, T \} \rangle_4^{\emptyset})$
  - with $prop(\{r\#1, p\#2, q\#3\}) = \{(r1, [2]) = T)$
- First state leads to final state $\langle \emptyset, \{p\#2\}, true, T \rangle_4^{\emptyset}$
  (Propagation history inhibits application of `r1`)
- All derivations from second state lead to $\langle \emptyset, \{p\#5, q\#6\}, true, T \rangle_7^{\emptyset}$
  (No state consisting of `p` can be reached from here)

Consistency

Theorem (Consistency)

If P range-restricted, confluent program, then $\mathcal{P}, \mathcal{CT}$ consistent

- Does not mean logical meaning is intended meaning
- `p<=>true, p<=>false` not confluent, inconsistent logical reading
- `p<=>q, p<=>false` not confluent, consistent logical reading
- `p<=>q, q<=>false` confluent, consistent logical reading

Soundness and completeness

Theorem (Strong soundness and completeness)

P a well-behaved program, C, C', C'', G goals.
Then the following statements are equivalent

a) $\mathcal{P}, \mathcal{CT} \models \forall\,(C \leftrightarrow G)$.

b) *G has a computation with answer C' such that* $\mathcal{P}, \mathcal{CT} \models \forall\,(C \leftrightarrow C')$.

c) *Every computation of G has an answer C'' such that $C' \equiv C''$ and* $\mathcal{P}, \mathcal{CT} \models \forall\,(C \leftrightarrow C'')$.

- Restriction to terminating: every computation finite
- Restriction to confluence: All computations for one goal lead to equivalent states

Soundness and completeness of failure

Theorem

P a well-behaved program, G a data-sufficient goal.

Then the following statements are equivalent

a) $\mathcal{P}, \mathcal{CT} \models \neg\exists G$

b) *G has a failed computation.*

c) *Every computation of G is failed.*

Example (I)

Example

```
p<=>q.
p<=>false.
q<=>false.
```

- Program is well-behaved (terminating, confluent)
- Goal $\mathtt{p}$ is data-sufficient
- $\mathcal{P}, \mathcal{CT} \models \neg\exists\mathtt{p}$ and every computation of $\mathtt{p}$ is failed

Example (II)

Example

```
p<=>q.
p<=>false.
```

- Program is terminating, but not confluent
- Goal $\mathtt{p}$ is data-sufficient
- $\mathcal{P}, \mathcal{CT} \models \neg\exists \mathtt{p}$ but not every computation of $\mathtt{p}$ is finitely failed
  - First rule gives successful answer $\mathtt{q}$

Example (III)

Example

```
p<=>p.
p<=>false.
```

- Program is not terminating, but confluent
- Goal $\mathtt{p}$ is data-sufficient
- $\mathcal{P}, \mathcal{CT} \models \neg\exists\mathtt{p}$ but not every computation of $\mathtt{p}$ is finitely failed
  - With only first rule computation is nonterminating

Completion

- Completion: adding rules until program becomes confluent
- Generate rules from critical pairs
- Generally propagation and simplification rule needed
- Completion not always possible
  - Newly added rule may introduce new critical pairs
  - Can lead to nonterminating process

Completion algorithm (I)

- Algorithm specified by set of inference rules
- Function *orient* generates propagation and simplification rule for critical pair (based on given termination order)
- Function *orient* is partial (does not apply if rules cannot be generated)

Completion algorithm (II)

Definition (*orient* function)

$\gg$ termination order, $(E_i \wedge C_i \ , \ E_j \wedge C_j)$ non-joinable critical pair, E_i, E_j CHR constraints and C_i, C_j built-in constraints.

Partial function $\mathit{orient}_{\gg}$ applies to $\{E_1 \wedge C_1 \ , \ E_2 \wedge C_2\}$ if

- $E_1 \wedge C_1 \ \gg \ E_2 \wedge C_2$ and
- E_1 is a nonempty conjunction and
- E_2 is a nonempty conjunction or $\mathcal{CT} \models C_2 \rightarrow C_1$

It returns rules

$$\{E_1 \Leftrightarrow C_1 \mid E_2 \wedge C_2, \quad E_2 \Rightarrow C_2 \mid C_1\}$$

where propagation rule generated only if $\mathcal{CT} \not\models C_2 \rightarrow C_1$

Completion algorithm (III)

- Condition chosen so that no rules generated if
  - states in c.p. cannot be ordered
  - empty headed rules would be generated
- Propagation rule $E_2 \Rightarrow C_2 \mid C_1$ ensures built-ins of both states are enforced
- No redundant propagation rules added ($\mathcal{CT} \models C_2 \rightarrow C_1$)

Completion algorithm (IV)

- Completion algorithm maintains set of c.p. and set of rules
- Start with program, set of nonjoinable critical pairs (P, S)
- Apply inference rules to exhaustion

Definition (Completion algorithm (I))

Simplification:

If $S_1 \mapsto S_1'$ then $(C \cup \{\{S_1, S_2\}\}, P) \longmapsto (C \cup \{\{S_1', S_2\}\}, P)$

Deletion:

If S_1 and S_2 are joinable then $(C \cup \{\{S_1, S_2\}\}, P) \longmapsto (C, P)$

- `Simplification`: replace state in c.p. by successor (leads to final states)
- `Deletion`: Removes joinable critical pairs

Completion algorithm (V)

Definition (Completion algorithm (II))

Orientation:

If $orient_{\gg}(\{S_1, S_2\}) = R$ then $(C \cup \{\{S_1, S_2\}\}, P) \longmapsto (C, P \cup R)$

Introduction:

If (S_1, S_2) is a c.p. of P not in C then $(C, P) \longmapsto (C \cup \{\{S_1, S_2\}\}, P)$

- `Orientation`: removes nonjoinable critical pair, adds new rules to P (if computed by $orient$)
- `Introduction`: Computes new critical pairs with new rules

Completion algorithm (VI)

- Completion succeeds when final state $(\emptyset, P')$ is reached
- Completion fails if nonjoinable c.p. cannot be oriented
  - States cannot be ordered
    - Termination order may be to blame
  - States consist of different built-ins only
    - Program has inconsistent logical reading
- Completion does not terminate if new rules always produce new c.p.

Examples (I)

Example

$\mathtt{p} \Leftrightarrow \mathtt{q}.$

$\mathtt{p} \Leftrightarrow \mathit{false}.$

- $\mathtt{p}$ leads to the c.p. $(\mathtt{q}, \mathit{false})$
- Simplification and Deletion do not apply
- Orientation adds rule $\mathtt{q} \Leftrightarrow \mathit{false}$
- No propagation rule ($\mathcal{CT} \models \mathit{false} \rightarrow \mathit{true}$)
- Introduce does not apply (no new overlaps)
- Completion succeeds, program confluent

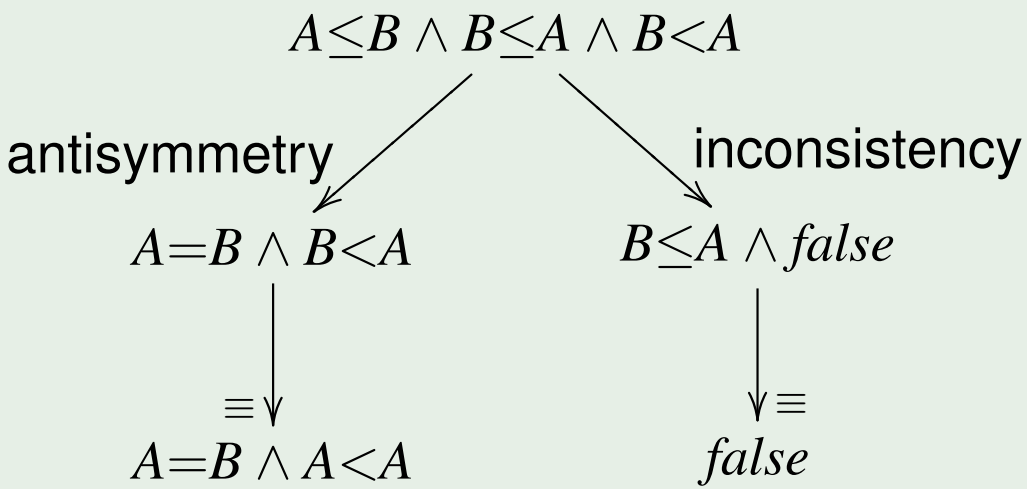
Examples (II)

Example (Partial order constraint extended (I))

Introduce $<$ constraint by adding one rule for inconsistency:

(inconsistency) $\quad X \leq Y \wedge Y < X \quad \Leftrightarrow \quad \mathit{false}$

Program not confluent (overlap with antisymmetry)



Examples (III)

Example (Partial order constraint extended (II))

Completion derives the rule

$$X{<}X \Leftrightarrow X{=}Y \mid \mathit{false},$$

which can be simplified to the rule

$$X{<}X \Leftrightarrow \mathit{false},$$

$\Rightarrow$ Discovery of irreflexivity of $<$

Examples (IV)

Example (Minimum and partial order constraint (I))

```
min1 @ min(X,X,Z)  ⇔  X=Z.
min2 @ min(X,Y,X)  ⇔  X leq Y.
min3 @ min(X,Y,Z) ∧ min(X,Y,Z1)  ⇔  min(X,Y,Z) ∧ Z=Z1.
```

- Termination order where $\texttt{min} \gg \texttt{leq}$
- Overlap of rule `min1` and `min2`:

  $\texttt{min(X,X,Z)} \wedge \texttt{X=Y} \wedge \texttt{X=Z}$

- Critical pair:

  $$(\texttt{X leq X}, \texttt{X=X})$$

- Becomes joinable with rule `r1 @ X leq X ⇔ true`

Examples (V)

Example (Minimum and partial order constraint (II))

```
min1 @ min(X,X,Z)  ⇔  X=Z.
min2 @ min(X,Y,X)  ⇔  X leq Y.
min3 @ min(X,Y,Z) ∧ min(X,Y,Z1)  ⇔  min(X,Y,Z) ∧ Z=Z1.
```

- Critical pairs of overlap between `min1` and `min3` joinable
- Critical pair from `min2` and first head constraint of `min3`:
  $(\texttt{min(X,Y,X)} \wedge \texttt{X=Z}, \texttt{X leq Y} \wedge \texttt{min(X,Y,Z)})$
- Critical pair joined with rule `r2`:
  `r2 @ X leq Y ∧ min(X,Y,Z) ⇔ X leq Y ∧ X=Z.`
- Also joins c.p. from `min2` and second head constraint of `min3`

Examples (VI)

Example (Minimum and partial order constraint (III))

```
min2 @ min(X,Y,X) ⇔ X leq Y.
min3 @ min(X,Y,Z) ∧ min(X,Y,Z1) ⇔ min(X,Y,Z) ∧ Z=Z1.
r2 @ X leq Y ∧ min(X,Y,Z) ⇔ X leq Y ∧ X=Z
```

- Critical pair of overlap between `min2` and `r2` joined by `r3 @ X leq Y ∧ X leq Y ⇔ X leq Y.`
- New rules `r1`, `r2`, `r3` reveal properties of `leq` and `min`
- Program with rules `min1`, `min2`, `min3` and `r1`, `r2`, `r3` is confluent and terminating

Examples (VII)

Example

$\geq$ and $\leq$ built-ins, $p \gg r \gg q$

```
r1 @ p(X,Y) ⇔ X ≥ Y ∧ q(X,Y).
r2 @ p(X,Y) ⇔ X ≤ Y ∧ r(X,Y).
```

- Critical pair from `r1` and `r2` not joinable

$$(\mathtt{X} \geq \mathtt{Y} \wedge \mathtt{q}(\mathtt{X},\mathtt{Y}),\ \mathtt{X} \leq \mathtt{Y} \wedge \mathtt{r}(\mathtt{X},\mathtt{Y}))$$

- Completion inserts two rules

```
r3 @ r(X,Y) ⇔ X ≤ Y | q(X,Y) ∧ X ≥ Y.
r4 @ q(X,Y) ⇒ X ≥ Y | X ≤ Y.
```

Examples (VIII)

Computations show that propagation rule `r4` is needed

Example computation (`r2`)

$$
\begin{array}{ll}
 & \mathtt{p}(\mathtt{X},\mathtt{Y}) \\
\mapsto_{Apply\ r2} & \mathtt{r}(\mathtt{X},\mathtt{Y}) \wedge \mathtt{X} \leq \mathtt{Y} \\
\mapsto_{Apply\ r3} & \mathtt{q}(\mathtt{X},\mathtt{Y}) \wedge \mathtt{X} = \mathtt{Y} \\
\mapsto_{Apply\ r4} & \mathtt{q}(\mathtt{X},\mathtt{Y}) \wedge \mathtt{X} = \mathtt{Y}
\end{array}
$$

Example computation (`r1`)

$$
\begin{array}{ll}
 & \mathtt{p}(\mathtt{X},\mathtt{Y}) \\
\mapsto_{Apply\ r1} & \mathtt{q}(\mathtt{X},\mathtt{Y}) \wedge \mathtt{X} \geq \mathtt{Y} \\
\mapsto_{Apply\ r4} & \mathtt{q}(\mathtt{X},\mathtt{Y}) \wedge \mathtt{X} = \mathtt{Y}
\end{array}
$$

Examples (IX)

Example

$\texttt{r1 @ p(X,Y)} \Leftrightarrow \texttt{X} \geq \texttt{Y} \wedge \texttt{q(X,Y)}.$

$\texttt{r2 @ p(X,Y)} \Leftrightarrow \texttt{X} \leq \texttt{Y} \wedge \texttt{q(Y,X)}.$

- Program not confluent, c.p. from `r1` and `r2` not joinable:

$$(\texttt{X} \geq \texttt{Y} \wedge \texttt{q(X,Y)}, \quad \texttt{X} \leq \texttt{Y} \wedge \texttt{q(Y,X)})$$

- No termination order for this c.p., rule would be

$\texttt{rf @ q(Y,X)} \Leftrightarrow \texttt{X} \leq \texttt{Y} \mid \texttt{q(X,Y)} \wedge \texttt{X} \geq \texttt{Y}.$

Correctness (I)

- When completion algorithm terminates successfully returned program confluent and terminating
  - Has same meaning as original program
- For proof of correctness
  - Rules have no local variables (range-restrictedness)
  - Completion could put existentially quantified variable in head of rule
  - Variables in head usually universally quantified

Correctness (II)

Theorem

P a range-restricted CHR program terminating with respect to a termination order $\gg$, C the set of non-joinable critical pairs of P.

If, for inputs (C, P) and $\gg$, completion succeeds with $(\emptyset, P')$, then program P' is

a) *terminating with respect to $\gg$,*

b) *confluent, and*

c) *logically equivalent to P.*

Example

Example (Not range-restricted program)

$\mathtt{p} \Leftrightarrow \mathtt{q(X)}.$

$\mathtt{p} \Leftrightarrow \mathit{true}.$

- Assume $\mathtt{q}$ does not hold for all possible values.
- Critical pair ($\mathtt{q(X)}$, true)
- Only way of orienting results in rule $\mathtt{q(X)} \Rightarrow \mathit{true}$
- Contradicts logical reading since $\mathtt{q}$ does not hold for all values

Failing completion and inconsistency

Failing completion can exhibit inconsistency in program

Theorem

P a CHR program, $\mathcal{CT}$ a complete theory for the built-in constraints. If completion fails and a remaining non-joinable critical pair consists only of built-in constraints that are not logically equivalent, then the logical meaning of P is inconsistent

- Most simple example $\{\mathtt{p} \Leftrightarrow \mathit{true}, \mathtt{p} \Leftrightarrow \mathit{false}\}$

Examples (I)

Example

$\texttt{p(X)} \Leftrightarrow \texttt{q(X).}$

$\texttt{p(X)} \Leftrightarrow \mathit{true}\texttt{.}$

$\texttt{q(X)} \Leftrightarrow \texttt{X>0.}$

- Critical pair ($\texttt{X>0}$, true)
- Cannot be oriented (only different built-ins)
- Completion fails, theorem indicates program is inconsistent

Examples (II)

Maximum program, has typo in second rule (`Y` should be `Z`)

Example (Maximum with typo)

```
r1 @ max(X,Y,Z) ⇔ X ≤ Y | Z = Y.
r2 @ max(X,Y,Z) ⇔ Y ≤ X | Y = X.
```

- Critical pair from `r1`, `r2`: not joinable and completion fails

$$(\mathtt{Z} = \mathtt{Y} \wedge \mathtt{X} \leq \mathtt{Y} \wedge \mathtt{Y} \leq \mathtt{X},\ \ \mathtt{Y} = \mathtt{X} \wedge \mathtt{X} \leq \mathtt{Y} \wedge \mathtt{Y} \leq \mathtt{X})$$

Examples (III)

Example (Maximum with typo continued)

Logical meaning together with theory for $\leq$ and $=$ is

$$\forall\ \texttt{X,Y,Z}\ (\texttt{X} \leq \texttt{Y} \rightarrow (\texttt{max(X,Y,Z)} \leftrightarrow \texttt{Z} = \texttt{Y}))$$
$$\forall\ \texttt{X,Y,Z}\ (\texttt{Y} \leq \texttt{X} \rightarrow (\texttt{max(X,Y,Z)} \leftrightarrow \texttt{Y} = \texttt{X}))$$

Not a consistent theory

- `max(1,1,0)` logically equivalent to `0=1` (first formula)
- `max(1,1,0)` also logically equivalent to `1=1` (second formula)
- Results in $\mathcal{P}, \mathcal{CT} \models \mathit{false} \leftrightarrow \mathit{true}$

Program specialization by completion (I)

Using completion to specialize programs and constraints

Example (Defining $<$)

Defining $<$ as special case of $\leq$ with usual rules, where $\neq$ is built-in

$$\texttt{r5 @ X} \leq \texttt{Y} \Leftrightarrow \texttt{X} \neq \texttt{Y} \mid \texttt{X} < \texttt{Y}.$$

- Program loses confluence
- With termination order $(\leq) \gg (<)$ completion inserts

$$\texttt{r6 @ X} < \texttt{Y} \wedge \texttt{Y} < \texttt{X} \Leftrightarrow \texttt{X} \neq \texttt{Y} \mid \mathit{false}.$$
$$\texttt{r7 @ X} < \texttt{Y} \wedge \texttt{X} < \texttt{Y} \Leftrightarrow \texttt{X} \neq \texttt{Y} \mid \texttt{X} < \texttt{Y}.$$

Program specialization by completion (II)

Example (Defining $<$ continued)

- Rule `r6` from critical pair of `r2` and `r5`

$$(\mathtt{X} = \mathtt{Y} \wedge \mathtt{X} \neq \mathtt{Y}, \quad \mathtt{X} < \mathtt{Y} \wedge \mathtt{Y} \leq \mathtt{X} \wedge \mathtt{X} \neq \mathtt{Y}).$$

- Rule `r7` from critical pair of `r4` and `r5`

$$(\mathtt{X} \leq \mathtt{Y} \wedge \mathtt{X} \neq \mathtt{Y}, \quad \mathtt{X} < \mathtt{Y} \wedge \mathtt{Y} \leq \mathtt{X} \wedge \mathtt{X} \neq \mathtt{Y}).$$

- `r6` implements antisymmetry of $<$
- `r7` implements impotence of conjunction by duplicate removal

Program specialization by completion (III)

Example (Append)

```
r1 @ append([],L,L) ⇔ true.
r2 @ append([X|L1],Y,[X|L2]) ⇔ append(L1,Y,L2).
```

No critical pairs, program is confluent
Adding rule for special case makes program nonconfluent

```
r3 @ append(L1,[],L3) ⇔ new(L1,L3),
```

Completion generates program for `new`:

```
r4 @ new([],[]) ⇔ true.
r5 @ new([A|B],[A|C]) ⇔ new(B,C).
```

Program specialization by completion (IV)

Example (Member predicate)

r1 @ member(X,[]) $\Leftrightarrow$ *false*.

r2 @ member(X,[H|T]) $\Leftrightarrow$ X = H | *true*.

r3 @ member(X,[H|T]) $\Leftrightarrow$ X $\neq$ H | member(X,T).

- In CHR query member(X,[1,2,3]) delays
- Prolog computes answers X=1, X=2, X=3

Program specialization by completion (V)

Example (Member predicate continued)

- If rule added `r4` added, program loses confluence

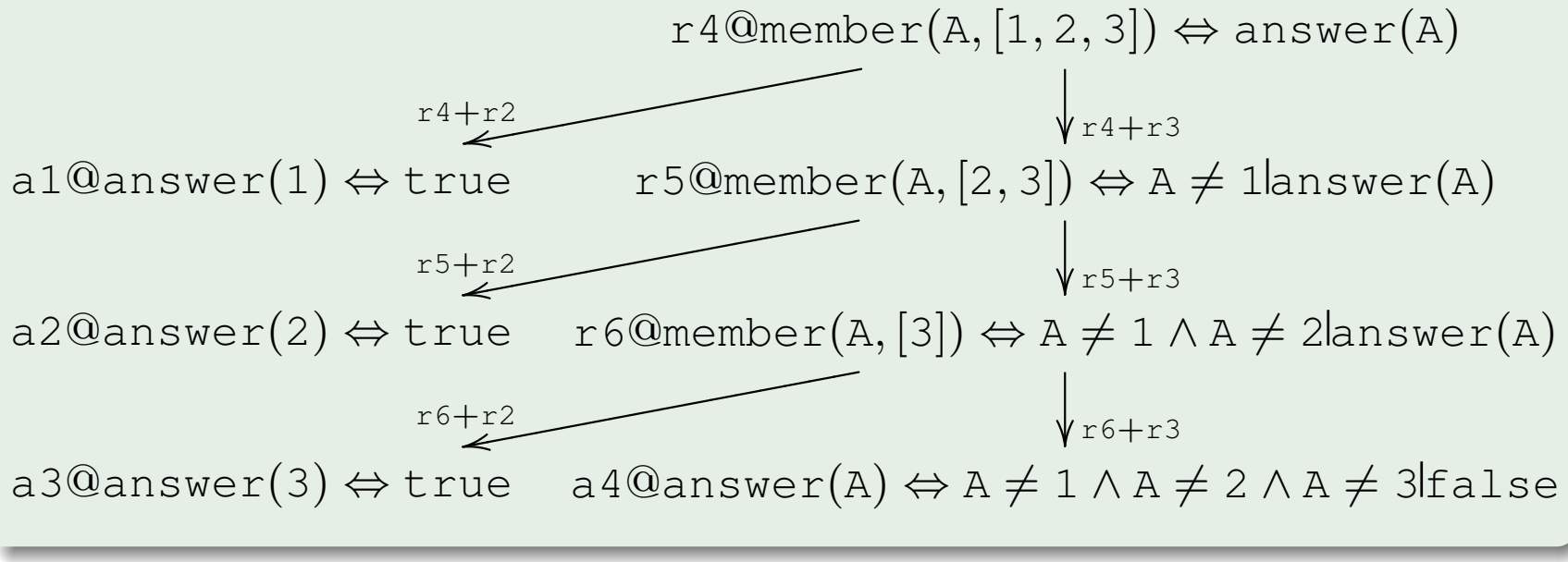
  `r4 @ member(X,[1,2,3])` $\Leftrightarrow$ `answer(X),`

- Completion generates rules equivalent to Prolog answers

  `a1 @ answer(1)` $\Leftrightarrow$ *true*.
  `a2 @ answer(2)` $\Leftrightarrow$ *true*.
  `a3 @ answer(3)` $\Leftrightarrow$ *true*.
  `a4 @ answer(X)` $\Leftrightarrow$ `X` $\neq$ `1` $\wedge$ `X` $\neq$ `2` $\wedge$ `X` $\neq$ `3` | *false*.

Program specialization by completion (VI)

Example (Member predicate rule generation)



Program specialization by completion (VII)

Example

- In `append` program query `append(X,[b|Y],[a,b,c|Z])` delays
- Prolog generates infinitely many answers

```
X = [a], Y = [c|Z])
X = [a,b,c], Z = [b|Y]
X = [a,b,c,X1], Z = [X1,b|Y]
X = [a,b,c,X1,X2], Z = [X1,X2,b|Y] ...
```

Program specialization by completion (VIII)

Example

Applying completion to two rules of append and

```
r3 @ append(X,[b|Y],[a,b,c|Z]) ⇔ answer(X,Y,Z).
```

results in

```
a1 @ answer([a],[c|Z],Z) ⇔ true.
a2 @ answer([a,b,c],Y,[b|Y]) ⇔ true.
a3 @ answer([a,b,c,H|L],Y,[H|L2])
                ⇔ answer([a,b,c|L],Y,L2).
```

- Rule `a1`: answer `X = [a], Y = [c|Z]`
- Rule `a2`: second answer `X = [a,b,c], Z = [b|Y]`
- Rule `a3`: remaining infinitely many Prolog answers

Modularity of termination and confluence

- Two ways of combining programs
  - Merging, taking the union of all rules
  - Hierarchically using modules, turning CHR constraints into built-ins of other program
- In abstract semantics any computation possible in one program also possible in merged program
- Modularity: property of program is preserved under union
- Union denoted by $\cup$

Examples

Example

First program $\{\mathtt{a} \Leftrightarrow \mathtt{b}\}$, second program $\{\mathtt{b} \Leftrightarrow \mathtt{a}\}$

- Both programs terminating
- Union $\{\mathtt{a} \Leftrightarrow \mathtt{b}\}, \{\mathtt{b} \Leftrightarrow \mathtt{a}\}$ not terminating

Example

First program $\{\mathtt{a} \Leftrightarrow \mathtt{b}\}$, second program $\{\mathtt{a} \Leftrightarrow \mathtt{c}\}$

- Both programs confluent
- Union $\{\mathtt{a} \Leftrightarrow \mathtt{b}\}, \{\mathtt{a} \Leftrightarrow \mathtt{c}\}$ is terminating but not confluent

$\Rightarrow$ Well-behavedness not preserved under union

Modular classes of CHR programs (I)

Definition

P, P_1, P_2 CHR programs

- c is **constraint of a program** P if its constraint symbol defined in P or if it is a built-in occurring in P (not CHR constraints only used in P)
- P_1, P_2 **nonoverlapping** if they have no defined CHR constraints in common
- P_1, P_2 **circular** if P_1 defines CHR constraint used in P_2 and vice versa
- Given goal, variable P_1**,** P_2**-shared** if it occurs in constraints of P_1 and constraints of P_2

Modular classes of CHR programs (II)

- Syntactic class of programs for preserving termination hard to find
- Union of noncircular, terminating programs is terminating for certain queries
- Union of noncircular, nonoverlapping programs is always confluent

Modularity of termination

- Termination nonmodular (circular definitions, shared variables)
- Common CHR symbols can be used by noncircular programs (not defined and used by both programs)
  - In at least one program all used CHR constraints are not defined in other
  - CHR constraint defined in both programs only defined recursively in one

Examples (I)

Example

$P1$: `c(f(X)) ⇔ X=g(Y) ∧ c(Y).`

$P2$: `d(g(Y)) ⇔ Y=f(Z) ∧ d(Z).`

- Any (finite) goal terminates in both programs
- Goal `c(f(X)) ∧ d(X)` does not terminate in union (shared variables, common function symbols)

  `c(f(X)) ∧ d(X)` $\mapsto_{P_1}$

  `X=g(Y) ∧ c(Y) ∧ d(g(Y))` $\mapsto_{P_2}$

  `X=g(f(W)) ∧ Y=f(W) ∧ c(f(W)) ∧ d(W)` $\mapsto_{P_1} \ldots$

$\Rightarrow$ Even noncircular definitions can lead to nontermination

Examples (II)

Example

Previous example, common function symbols replaced by built-ins

$P1$: `c(FX)` $\Leftrightarrow$ `f1(FX,X)` | `g1(X,Y)` $\wedge$ `c(Y)`.
$P2$: `d(GY)` $\Leftrightarrow$ `g2(GY,Y)` | `f2(Y,Z)` $\wedge$ `d(Z)`.

- $f1(X,Y)$ and $f2(X,Y)$ both defined as $X = f(Y)$
- $g1(X,Y)$ and $g2(X,Y)$ both defined as $X = g(Y)$
- `c(FX)` $\wedge$ `f1(FX,X)` $\wedge$ `d(X)` not terminating in union
- Built-in constraint from one program implies guard constraint from other and vice versa (hard to rule out)

Examples (III)

Example

$P1$: `c(X,N)` $\Leftrightarrow$ `f(X,N)` | `g(X,N)` $\wedge$ `c(X,N+1).`

$P2$: `d(Y,N)` $\Leftrightarrow$ `g(Y,N)` | `f(Y,N+1)` $\wedge$ `d(Y,N+1).`

- $f(X,N)$ defined as $X \; mod \; 2^N{=}0 \; \wedge \; X{>}N \; \wedge \; N{>}0$
- $g(X,N)$ defined as $X \; mod \; 3^N{=}0 \; \wedge \; X{>}N \; \wedge \; N{>}0$
- Constraints f and g never imply each other
- Goal `c(X,N)` $\wedge$ `f(X,N)` $\wedge$ `d(X,N)` not terminating in union

$\Rightarrow$ Nontermination problem persists over P_1,P_2-shared variables

Modularity of termination

- Common symbols influences termination of union
  - Circularity in programs via shared variables
- Restricting domain of P_1,P_2-shared variables to be finite makes termination modular for union of noncircular programs

Theorem

P_1, P_2 well-behaved programs. If P_1, P_2 noncircular and P_1,P_2-shared variables in query defined over finite domains only then $P_1 \cup P_2$ is terminating

Example (I)

Example

$P1$: `c(X,N)` $\Leftrightarrow$ `f(X,N)` | `g(X,N)` $\wedge$ `c(X,N+1).`

$P2$: `d(Y,N)` $\Leftrightarrow$ `g(Y,N)` | `f(Y,N+1)` $\wedge$ `d(Y,N+1).`

- $f(X,N)$ defined as $X \; mod \; 2^N{=}0 \; \wedge \; X{>}N \; \wedge \; N{>}0$
- $g(X,N)$ defined as $X \; mod \; 3^N{=}0 \; \wedge \; X{>}N \; \wedge \; N{>}0$
- Programs noncircular, `X` and `N` P_1,P_2-shared
- `c(X,N)`$\wedge$ `f(X,N)`$\wedge$`d(X,N)` not terminating

Example (II)

Example (continued)

Finite domain constraint `X in D`: `X` takes values from given finite list `D`

Adding finite domain constraints in query

```
X in [2,4,6,8]∧ N in [1,2] ∧ c(X,N)∧ f(X,N)∧d(X,N)
```

leads to state

```
X in [6] ∧ N in [1] ∧ g(X,N)∧f(X,N)∧d(X,N)∧c(X,N+1)
```

Next state contains `f(X,N+1)` but `f(6,2)` does not hold

$\Rightarrow$ computation fails

Modularity of confluence

Theorem

P_1, P_2 well-behaved and nonoverlapping, $P_1 \cup P_2$ terminating, then $P_1 \cup P_2$ is confluent

Example

Union of well-behaved programs $\{\mathtt{a} \Leftrightarrow \mathtt{b}\}$, $\{\mathtt{b} \Leftrightarrow \mathtt{c}\}$ is confluent (programs nonoverlapping)

- If union terminating, confluence test can be used
- Only c.p. from rules in different programs interesting
- Confluence test can be made incremental

Example

Nonconfluent union $\{\mathtt{a} \Leftrightarrow \mathtt{b},\ \mathtt{a} \Leftrightarrow \mathtt{c}\}$ (both defining $\mathtt{a}$)

Operational equivalence

- Operational equivalence fundamental question in programming language semantics
- Correctness of program transformation needs notion of equivalence
- In CHR: For combining constraints solvers
- Operational equivalence: For any given query, both programs lead to same answer
- In CHR decidable, sufficient, and necessary condition for well-behaved programs

Operational equivalence of programs (I)

Definition (Operational equivalence)

State S is P_1,P_2-joinable iff $S \mapsto_{P_1}^{+} S_1$ and $S \mapsto_{P_2}^{+} S_2$ such that $S_1 \equiv S_2$ or S is final state in both programs.

P_1 and P_2 operationally equivalent if all states are P_1,P_2-joinable.

Operational equivalence of programs (II)

- Test for operational equivalence of well-behaved programs:
  - Execute minimal states as queries in both programs
  - Programs operational equivalent if equivalent states reached

Theorem

Well-behaved programs P_1, P_2 operationally equivalent iff all minimal states of rules in P_1, P_2 are P_1,P_2-joinable

Example

Example (Two programs for `max`)

`max(X,Y,Z)` $\Leftrightarrow$ `X`$\leq$`Y` | `Z=Y.`

`max(X,Y,Z)` $\Leftrightarrow$ `Y<X` | `Z=X.`

`max(X,Y,Z)` $\Leftrightarrow$ `X<Y` | `Z=Y.`

`max(X,Y,Z)` $\Leftrightarrow$ `Y`$\leq$`X` | `Z=X.`

- `max(X,Y,Z)`$\wedge$`X`$\leq$`Y` shows operational nonequivalence
- Can reduce to `Z` = `Y` in first program, is final state in second program
- Programs are logically equivalent

Operational equivalence of constraints

- Notion of operational equivalence can be too strict
- Operational equivalence of constraints in different programs
- Decidable sufficient syntactic condition for well-behaved programs
- Only sufficient but not necessary condition

Definition

Definition (Operational c-equivalence)

c a CHR constraint symbol. A c-state is a state where all CHR constraints have the symbol c.

c a CHR constraint defined in P_1 and P_2. P_1 and P_2 operationally c-equivalent if all c-states P_1,P_2-joinable

Example

Example

Let P_1 be the program:

`p(a)` $\Leftrightarrow$ `s.`
`p(b)` $\Leftrightarrow$ `r.`
`s`$\wedge$`r` $\Leftrightarrow$ *true*.

Program P_2 consists of only the first two rules.

- Considering only `p`, goals `p(a)`, `p(b)` not sufficient for operational `p`-equivalence
- in P_1 `p(a)` $\wedge$ `p(b)` leads to *true*, in P_2 to `s` $\wedge$ `r`
- Including minimal states for `s` and `r` (`s` $\wedge$ `r`) leads to different program behavior

Dependency

Definition (Dependency)

- CHR constraint (symbol) c directly depends on d if there is a rule defining c and using d
- Dependency relation is reflexive transitive closure of direct dependency
- Given P_1 an P_2, c-dependent constraint is constraint depending on c in P_1 and P_2

Theorem for operational equivalence

Definition

c-minimal states are minimal states of programs P_1, P_2 that only contain c-dependent CHR constraints

Theorem

c a CHR constraint defined in well-behaved programs P_1, P_2.
If all c-minimal states are P_1,P_2-joinable then P_1, P_2 are operationally c-equivalent

Examples (I)

Example (Sum)

`sum(List,Sum)` holds if `Sum` is the sum of elements in `List`

Program P_1:

```
sum([],Sum) ⇔ Sum=0.
sum([X|Xs],Sum) ⇔  sum(Xs,Sum1) ∧ Sum=Sum1+X.
```

Program P_2:

```
sum([],Sum) ⇔ Sum=0.
sum([X|Xs],Sum) ⇔ sum1(X,Xs,Sum).
 sum1(X,[],Sum) ⇔ Sum=X.
 sum1(X,Xs,Sum) ⇔ sum(Xs,Sum1) ∧ Sum=Sum1+X.
```

Examples (II)

Example (Sum continued)

`sum`-minimal states are `sum([],Sum)` and `sum([X|Xs],Sum)`

- For `sum([],Sum)` final state `Sum=0` in P_1 and P_2
- Computation for `sum([X|Xs],Sum)` in P_1:
  `sum([X|Xs],Sum)` $\mapsto_{P_1}$ `sum(Xs,Sum1)` $\wedge$ `Sum=Sum1+X`
- Computation for `sum([X|Xs],Sum)` in P_2:
  `sum([X|Xs],Sum)` $\mapsto_{P_2}$ `sum1(X,Xs,Sum)` $\mapsto_{P_2}$
  `sum(Xs,Sum1)` $\wedge$ `Sum=Sum1+X`

All `sum`-minimal states P_1,P_2-joinable

$\Rightarrow$ P_1 and P_2 operationally `sum`-equivalent

Examples (III)

Example

Program P_1

```
p(X) ⇔  X>0 | q(X).
q(X) ⇔  X<0 | true.
```

Program P_2

```
p(X) ⇔  X>0 | q(X).
q(X) ⇔  X<0 | false.
```

P_1, P_2 operationally `p`-equivalent, but `p`-minimal state `q(X) ∧ X>0` is not P_1, P_2-joinable

Shows reason why only sufficient but no necessary condition for operational c-equivalence can be given

Examples (IV)

Example

Program P_1

$\mathtt{p} \Leftrightarrow \mathtt{s}\,.$

$\mathtt{s} \wedge \mathtt{q} \Leftrightarrow \mathit{true}\,.$

Program P_2

$\mathtt{p} \Leftrightarrow \mathtt{s}\,.$

$\mathtt{s} \wedge \mathtt{q} \Leftrightarrow \mathit{false}\,.$

- $\mathtt{s}$ and $\mathtt{p}$ are $\mathtt{p}$-dependent
- $\mathtt{s}$ is only $\mathtt{s}$-dependent constraint, analogous for $\mathtt{q}$
- All $\mathtt{p}$-, $\mathtt{s}$-, $\mathtt{q}$-minimal states P_1,P_2-joinable but programs not operationally equivalent
- If notion of c-equivalence extended to sets, $\mathtt{p}$-, $\mathtt{s}$-, $\mathtt{q}$-minimal states include indicative state $\mathtt{s} \wedge \mathtt{q}$

Removal of redundant rules

- Union and completion may result in redundant rules
- Variation of operational equivalence to test redundancy
- Detects rules that can be removed without changing operational semantics

Definition (Redundancy)

$P \backslash r$ denotes program P without rule r.
Rule r is redundant in P iff for all states S

$$\text{if } S \mapsto_P^* S_1 \text{ then } S \mapsto_{P \backslash r}^* S_2 \text{ such that } S_1 \equiv S_2$$

Example

Example (Union of `max` programs)

```
r1 @ max(X,Y,Z) ⇔ X<Y | Z=Y.
r2 @ max(X,Y,Z) ⇔ X≥Y | Z=X.
r3 @ max(X,Y,Z) ⇔ X≤Y | Z=Y.
r4 @ max(X,Y,Z) ⇔ X>Y | Z=X.
```

`r3` can always be applied when `r1` can be applied with same answer (not vice versa) $\Rightarrow$ `r1` is redundant, analogously `r4`

Theorem for redundant rules

- Removing rule from well-behaved program can destroy confluence
- Equivalence test not directly applicable

Theorem

P be a well-behaved program. Rule r is redundant with respect to P iff $P\backslash r$ is well-behaved and all minimal states of P and $P\backslash r$ are $P, P\backslash r$-joinable

- Specialize equivalence test
  - Check if computation due to candidate rule for removal can be performed by $P\backslash r$
  - State in computation for minimal state of r must be reachable in $P\backslash r$

Examples (I)

Example (Union of `max` programs continued)

```
r1 @ max(X,Y,Z) ⇔ X<Y | Z=Y.
r2 @ max(X,Y,Z) ⇔ X≥Y | Z=X.
r3 @ max(X,Y,Z) ⇔ X≤Y | Z=Y.
r4 @ max(X,Y,Z) ⇔ X>Y | Z=X.
```

- Any subset of program still well-behaved
- Removal of rule `r1` (min. state `max(X,Y,Z)∧X<Y`), run

  P: `max(X,Y,Z)∧X<Y` $\mapsto$ `X<Y∧Z=Y` by rule `r1`

  $P\backslash\{$`r1`$\}$: `max(X,Y,Z)∧X<Y` $\mapsto$ `X<Y ∧ Z=Y` by rule `r3`
- `r3` enables same computation $\Rightarrow$ `r1` redundant
- Redundancy of `r4` shown analogously

Examples (II)

Example (Union of `max` programs continued)

```
r1 @ max(X,Y,Z) ⇔ X<Y | Z=Y.
r2 @ max(X,Y,Z) ⇔ X≥Y | Z=X.
r3 @ max(X,Y,Z) ⇔ X≤Y | Z=Y.
r4 @ max(X,Y,Z) ⇔ X>Y | Z=X.
```

- Rule `r2` not redundant

  P: `max(X,Y,Z)`$\wedge$`X≥Y` $\mapsto$ `X≥Y` $\wedge$ `Z=X` by rule `r2`

  $P\backslash\{$`r2`$\}$: `max(X,Y,Z)`$\wedge$`X≥Y` $\not\mapsto$

- Program without redundant rules consists of `r2` and `r3`

Examples (III)

Example (Strict order relation)

```
duplicate    @ X less Y \ X less Y ⇔ true.
irreflexivity@ X less X ⇔ false.
antisymmetry @ X less Y ∧ Y less X ⇔ false.
transitivity @ X less Y ∧ Y less Z ⇒ X less Z.
```

- `antisymmetry` is redundant (`transitivity` then `irreflexivity`)
- Other rules not redundant

Examples (IV)

Resulting program not necessarily unique

Example

`r1 @ p` $\Leftrightarrow$ `q.`

`r2 @ p` $\Leftrightarrow$ false `.`

`r3 @ q` $\Leftrightarrow$ false `.`

- Either `r1` or `r2` can be removed by redundancy removal
- Hence, program without redundant rules not unique

Worst-case time complexity (I)

- Semi-automatic time complexity analysis based on semi-naive implementations of CHR (abstract semantics)
- Better results through refined semantics and compiler optimizations in particular indexing
  - Usually head constraints connected through common variables
  - Search for partner constraints only where variables shared
  - Indexing on argument position of common variable

    $\Rightarrow$ Partner constraints often found in constant time

Worst-case time complexity (II)

- Run time based on number of rule applications and rule application attempts
- Meta-theorem for wc time complexity of simplification rule programs combines
  - Derivation length
  - Number and cost of rule tries
  - Cost of rule application
- Number of potential rule applications known from program text (given ranking)

Example (I)

Example (Complexity of `even` constraint)

`even(s(N)) ⇔ N=s(M)∧even(M).`

- Time complexity of single `even` linear in derivation length (rank)
- Time complexity of several ground `even` constraints also linear (where rank is sum of ranks of individual constraints)

Example (II)

Example (Complexity of `even` constraint continued)

- Adding second rule

  `even(s(N))` $\Leftrightarrow$ `N=s(M)`$\wedge$`even(M)` .

  `even(s(X))`$\wedge$`even(X)` $\Leftrightarrow$ *false* .

- New rule must be tried for all pairs of `even` constraints
- Must be tried after computation step with single `even` constraint
- Rule tries in derivation step at worst quadratic in number of constraints in query
- Rank of query is bound on number of constraints
- Number of derivation steps also bounded by rank of query
- Overall: implementation is cubic in rank of query

Simplification rules

Computational phases when rule is applied:

- **Head matching**
  - Find atomic CHR constraints in current state to match head of rule
- **Guard checking**
  - Check if current built-in constraints imply guard of rule under found matching
- **Body handling**
  - According to rule type remove matched constraints
  - Guard and body with built-in and CHR constraints added

Theorem for simplification rules

Theorem

r a simplification rule $H \Leftrightarrow G \mid C \wedge B$
(H conjunction of n CHR constraints, C, B built-ins, B CHR constraints)

A worst-case time complexity of applying r in state with c constraints is:

$$O(c^n(O_H + O_G) \ + \ (O_C + O_B)),$$

(Complexities:
O_H: head matching, O_G: guard checking, O_C: adding C to state, O_B removing matched head and adding B to state)

Theorem for programs

- Worst case complexity of rule application
  - Largest number of CHR constraints of any state in derivation bound by $O(c + D)$
  - Most costly rule to be tried and applied

Theorem

P containing only simplification rules, D worst-case derivation length of given query.

Then the worst-case time complexity of given query is

$$O(D \sum_i ((c + D)^{n_i} (O_{H_i} + O_{G_i}) \ + \ (O_{C_i} + O_{B_i})))$$

(i ranges over rules in P)

Complexity of programs

- Cost of rule tries dominates complexity of semi-naive implementation of CHR
- Often sufficient to consider worst rule for computing complexity measure

Typical complexities (I)

- Cost of syntactic matching O_H determined by syntactic size, thus quasi-constant
- Cost O_B (adding, removing) often constant
- Complexity of handling built-ins assumed not to depend on constraints accumulated so far
- Constant time for arithmetics, quasi-constant time for matching and unification assumed

Typical complexities (II)

- Complexity of guard checking O_G usually at most complexity of adding respective constraints
- Complexity of adding built-ins O_C often linear in their size
- In many cases D contains factor $c \Rightarrow c + D$ simplifies to D

$\Rightarrow$ simplified worst-case time complexity estimate

$$O(\sum_i (D^{n_i+1} O_{G_i} \ + \ D O_{C_i}))$$

Example

Example (One rule program with successor notation)

`c(s(X)) ⇔ c(X) ∧ c(X).`

- Removing successor doubles number of constraints
- Exponential ranking needed
  - $rank(c(t)) = 2^{size(t)} - 1$
  - $size(0) = 0$
  - $size(s(N)) = 1 + size(N)$
- Complexity exponential in size of argument of c ($n = size(t)$):

$$O(2^n((1+2^n)^1(1+0)+(0+1)) = O(2^n 2^n) = O(4^n)$$

- $O(2^n)$ derivation steps, $O(2^n)$ constraints in each state