Program analysis

Table of Contents

Program analysis
Termination
Confluence
Completion
Modularity of termination and confluence
Operational equivalence
Worst-case time complexity
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 $> \text{ on natural numbers used as basis of termination order}$
- 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 (Ranking)

Formula $B$: $\text{rank}(B) \geq 0$ required

Built-in $C$: $\text{rank}(C) = 0$ required

Conjunction: $\text{rank}((A \land B)) = \text{rank}(A) + \text{rank}(B)$

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

\[
\text{rank}(f(t_1, \ldots, t_n)) = a_0^f + a_1^f \times \text{rank}(t_1) + \ldots + a_n^f \times \text{rank}(t_n)
\]

$\text{rank}(s) > \text{rank}(t)$ is order constraint
Syntactic size

**Definition (Syntactic size)**

Syntactic size of term can be expressed as the ranking

\[
\text{size}(f(t_1, \ldots, t_n)) = 1 + \text{size}(t_1) + \ldots + \text{size}(t_n)
\]

**Example**

- \( \text{size}(f(a, g(b, c))) = 5 \)
- \( \text{size}(f(a, X)) = 2 + \text{size}(X) \) where \( \text{size}(X) \geq 0 \)
- \( \text{size}(f(g(X), X)) \geq \text{size}(a, X) \) because 
  \[ 2 + 2 \times \text{size}(X) \geq 2 + \text{size}(X) \]
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. $s = t \rightarrow \text{rank}(s) = \text{rank}(t)$)

- 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

\[ \forall (C \rightarrow \text{rank}(H) > \text{rank}(B)) \]

\text{(C conjunction of rule’s built-ins, rank CHR ranking function)}

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

\[ \text{even}(s(N)) \iff N = s(M) \land \text{even}(M). \]

- **even(N) and even(0) delay**
- **even(s(N)) results in** \(N = s(M), \text{even}(M)\)
- **even(s(0)) fails since** \(0 = s(M)\)

Suitable polynomial interpretation

\[ \text{rank}(\text{even}(N)) = \text{size}(N) \]

Resulting ranking condition (holds) for rule

\[ N = s(M) \rightarrow \text{rank}(s(N)) > \text{rank}(M) \]
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 $G$:* 

$$D(G) \leq \text{rank}(G)$$
Confluence

- Confluence guarantees that any computation for a goal results in the 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 $H'_1 \land H'_2 \land C$ is minimal state of rule $r$ and $r$ is applicable to state $S$ then there exists goal $G'$ such that $H'_1 \land H'_2 \land C \land G' \equiv S$ (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 $S_1 \rightarrow^* S'_1$ and $S_2 \rightarrow^* S'_2$ and $S'_1 \equiv S'_2$
Confluence diagram

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

  $S \xrightarrow{*} S_1, S \xrightarrow{*} S_2$ 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), 
\( H_i \land A_i \) conjunction of head constraints, \( C_i \) guard, \( B_i \) body. 
A (nontrivial) overlap \( S \) of rule \( R_1 \) and \( R_2 \) is 

\[
S = (H_1 \land A_1 \land H_2 \land (A_1 = A_2) \land C_1 \land C_2)
\]

if \( A_1 \) and \( A_2 \) nonempty and \( CT \models \exists ((A_1 = A_2) \land C_1 \land C_2) \)
Critical pairs

**Definition (Critical pair)**

\[ S_1 = (B_1 \land H_2 \land (A_1 = A_2) \land C_1 \land C_2) \]  
\[ S_2 = (H_1 \land B_2 \land (A_1 = A_2) \land C_1 \land C_2) \]

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 $S, S_1, S_2$

If $S \xrightarrow{\rightarrow} S_1, S \xrightarrow{\rightarrow} S_2$ then $S_1$ and $S_2$ 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

\[ p \Leftrightarrow q. \]
\[ p \Leftrightarrow false. \]

- One overlap \( p \)
- Critical pair \((q, false)\): final and different, thus nonjoinable final states

Example (Coin throw)

\[ \text{throw}(\text{Coin}) \Leftrightarrow \text{Coin} = \text{head}. \]
\[ \text{throw}(\text{Coin}) \Leftrightarrow \text{Coin} = \text{tail}. \]

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

Example

\( p(X) \land q(Y) \iff \text{true} \).

- Overlap of rule with itself: \( p(X) \land q(Y_1) \land q(Y_2) \)
- Critical pair \((q(Y_1), q(Y_2))\), \(Y_1\) and \(Y_2\) different variables from overlap
- Analogous situation for overlap \( p(X_1) \land p(X_2) \land q(Y) \)
- Nonjoinability does not arise for rule

\[ p(X) \land q(Y) \iff X=Y \mid \text{true} \].
Example (Destructive assignment)

assign(Var, New), cell(Var, Old) \iff 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)

\[
\begin{align*}
\text{max} (X, Y, Z) & \iff X \le Y \land Y = Z. \\
\text{max} (X, Y, Z) & \iff Y \le X \land X = Z.
\end{align*}
\]

- Only overlap: \( \text{max}(X, Y, Z) \land X \le Y \land Y \le X \)
- Critical pair: \((Y=Z \land X \le Y \land Y \le X, X=Z \land X \le Y \land Y \le X)\)
- Joinable: both states equivalent to \(X=Y \land Y=Z\)
Examples (V)

Example (Merge (I))

\[
\text{merge}([], L2, L3) \iff L2 = L3.
\]

\[
\text{merge}(L1, [], L3) \iff L1 = L3.
\]

- Eight critical pairs, some joinable, some not
- Critical pair from first two rules:
  \[
  ([]=L1 \land L2=[] \land L2=L3, \; []'=L1 \land L2=[] \land L1'=L3)
  \]
  - Joinable: Both states equivalent to \( L1=[] \land L2=[] \land L3=[] \)
Examples (V)

Example (Merge (II))

merge([X|R1],L2,L3) ⇔ L3=[X|R3] ∧ merge(R1,L2,R3).
merge(L1,[Y|R2],L3) ⇔ L3=[Y|R3] ∧ merge(L1,R2,R3).

▶ Critical pair from third and forth rule:

(L1=[X|R1] ∧ L2=[Y|R2] ∧ L3=[X|R3] ∧
merge(R1,L2,R3),
L1=[X|R1] ∧ L2=[Y|R2] ∧ L3=[Y|R3] ∧
merge(L1,R2,R3))

▶ E.g. query merge([a],[b],L) can result in two lists L
⇒ not confluent (order of elements in list L not determined)
Examples (VI)

Example (Partial order constraint (I))

- Reflexivity: $X \leq X \iff \text{true}$
- Antisymmetry: $X \leq Y \land Y \leq X \iff X = Y$

- Overlap of reflexivity and antisymmetry

- Critical pair joinable (multiplicity matters in CHR)
Examples (VII)

Example (Partial order constraint (II))

duplicate @ \( X \leq Y \land X \leq Y \iff X \leq Y \).

antisymmetry @ \( X \leq Y \land Y \leq X \iff X = Y \).

transitivity @ \( X \leq Y \land Y \leq Z \implies X \leq Z \).

- Overlap of antisymmetry and transitivity (left-most head constraint): \( X \leq Y \land Y \leq Z \land Y \leq X \)

- Critical pair: \((X \leq Y \land Y \leq X \land Y \leq Z \land X \leq Z, X = Y \land X \leq Z)\)

- Joinable (first state leads to second state):

  \[
  \begin{align*}
  & X \leq Y \land Y \leq X \land Y \leq Z \land X \leq Z \\
  & Y \leq Z \land X \leq Z \land X = Y \\
  & X \leq Z \land X = Y
  \end{align*}
  \]

  \( \Rightarrow \) apply antisymmetry

  \( \Rightarrow \) apply duplicate
Confluence test for abstract semantics (I)

Example (Joinable states)

\[ r_1 \land p \Rightarrow q. \]
\[ r_2 \land r \land q \Leftrightarrow \text{true}. \]
\[ r_3 \land r \land p \land q \Leftrightarrow s. \]
\[ r_4 \land s \Leftrightarrow p \land q. \]

Computation leads to final state \( p \land q \) no matter which rule applied.
Confluence test for abstract semantics (II)

Example (Nonjoinable states)

- No reapplication of $r_1$ possible to remove $p$ in left branch
- $r_1$ can be applied to $p \land q$ (but $q$ cannot be removed)
Consider propagation history for abstract semantics

1. 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 $S \land B$ with $\omega_t$ state $\langle \emptyset, S', B, prop(S') \rangle^n_V$
  - $S'$: numbered CHR constraints such that $S = \text{chr}(S')$
  - $V$: 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 \( r \land p \land q \)
- Associated \( \omega_t \) state \( \langle \emptyset, \{ r\#1, p\#2, q\#3 \}, \text{true}, \emptyset \rangle^0_4 \)
- Resulting c.p. \( (\langle \{ \text{true} \}, \{ p\#2 \}, \text{true}, T \rangle^0_4, \langle \{ s \}, \emptyset, \text{true}, T \rangle^0_4) \)
  - with \( \text{prop}(\{ r\#1, p\#2, q\#3 \}) = \{(r1, [2]) = T\) \)
- First state leads to final state \( \langle \emptyset, \{ p\#2 \}, \text{true}, T \rangle^0_4 \)
  (Propagation history inhibits application of \( r_1 \))
- All derivations from second state lead to \( \langle \emptyset, \{ p\#5, q\#6 \}, \text{true}, T \rangle^0_7 \)
  (No state consisting of \( p \) can be reached from here)
Consistency

Theorem (Consistency)

If $P$ range-restricted, confluent program, then $P, CT$ consistent

- Does not mean logical meaning is intended meaning
- $p \Leftrightarrow \text{true}, p \Leftrightarrow \text{false}$ not confluent, inconsistent logical reading
- $p \Leftrightarrow q, p \Leftrightarrow \text{false}$ not confluent, consistent logical reading
- $p \Leftrightarrow q, q \Leftrightarrow \text{false}$ confluent, consistent logical reading
Theorem (Strong soundness and completeness)

\[ P \text{ a well-behaved program, } C, C', C'', G \text{ goals.} \]

Then the following statements are equivalent

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

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

c) \( G \text{ has a computation with answer } C'' \) 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) $P, CT \models \neg \exists G$

b) $G$ has a failed computation.

c) Every computation of $G$ is failed.
Example (I)

Example

\[ p \iff q. \]

\[ p \iff \text{false}. \]

\[ q \iff \text{false}. \]

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

Example

\( p \iff q. \)

\( p \iff \text{false}. \)

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

Example

\[ p \leftrightarrow p. \]
\[ p \leftrightarrow \text{false}. \]

- Program is not terminating, but confluent
- Goal \( p \) is data-sufficient
  - 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 \( (orient \text{ function}) \)

\[
\triangleright \text{termination order, } (E_i \land C_i, E_j \land C_j) \text{ non-joinable critical pair, } E_i, E_j \\
\text{CHR constraints and } C_i, C_j \text{ built-in constraints.}
\]

Partial function \( orient \) applies to \( \{E_1 \land C_1, E_2 \land C_2\} \) if

- \( E_1 \land C_1 \triangleright E_2 \land C_2 \) and
- \( E_1 \) is a nonempty conjunction and
- \( E_2 \) is a nonempty conjunction or \( CT \models C_2 \rightarrow C_1 \)

It returns rules

\[
\{E_1 \Leftrightarrow C_1 \mid E_2 \land C_2, \quad E_2 \Rightarrow C_2 \mid C_1\}
\]

where propagation rule generated only if \( CT \not\models C_2 \rightarrow C_1 \)
Completion algorithm (III)

- Condition chosen so that no rules generated if states in c.p. cannot be ordered and 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 ($CT \models C_2 \rightarrow C_1$).
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 \(S_1 \rightarrow S'_1\) then \((C \cup \{\{S_1, S_2\}\}, P) \rightarrow (C \cup \{\{S'_1, S_2\}\}, P)\)

**Deletion:**
If \(S_1\) and \(S_2\) are joinable then \((C \cup \{\{S_1, S_2\}\}, P) \leftarrow (C, P)\)

- **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 $\text{orient} \supseteq \{S_1, S_2\} = R$ then $(C \cup \{\{S_1, S_2\}\}, P) \mapsto (C, P \cup R)$

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

- **Orientation**: removes nonjoinable critical pair, adds new rules to $P$ (if computed by $\text{orient}$)
- **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

\[ p \iff q. \]
\[ p \iff false. \]

- \( p \) leads to the c.p. \((q, \text{false})\)
- Simplification and Deletion do not apply
- Orientation adds rule \( q \iff false \)
- No propagation rule \((CT \models false \rightarrow true)\)
- 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:

\[(\text{inconsistency}) \quad X \leq Y \land Y < X \iff \text{false}\]

Program not confluent (overlap with antisymmetry)

\[
\begin{align*}
A \leq B \land B \leq A \land B < A & \quad \text{antisymmetry} \\
\equiv & \\
A = B \land A < A & \\
\equiv & \\
B \leq A \land \text{false} & \quad \text{inconsistency} \\
\equiv & \\
\text{false} &
\end{align*}
\]
Examples (III)

Example (Partial order constraint extended (II))

Completion derives the rule

\[ X < X \iff X = Y \mid false, \]

which can be simplified to the rule

\[ X < X \iff false, \]

\[ \Rightarrow \text{Discovery of irreflexivity of } < \]
Examples (IV)

Example (Minimum and partial order constraint (I))

\[
\begin{align*}
\text{min1} & @ \min(X, X, Z) \iff X = Z. \\
\text{min2} & @ \min(X, Y, X) \iff X \leq Y. \\
\text{min3} & @ \min(X, Y, Z) \land \min(X, Y, Z_1) \iff \min(X, Y, Z) \land Z = Z_1.
\end{align*}
\]

- **Termination order where** \( \min \succ \leq \)
- **Overlap of rule** \text{min1} \text{ and } \text{min2}:

  \[ \min(X, X, Z) \land X = Y \land X = Z \]

- **Critical pair:**

  \[(X \leq X, X = X)\]

- **Becomes joinable with rule** \( r_1 @ X \leq X \iff \text{true} \)
Examples (V)

Example (Minimum and partial order constraint (II))

\[ \text{min1 } @ \text{ min}(X,X,Z) \iff X=Z. \]
\[ \text{min2 } @ \text{ min}(X,Y,X) \iff X \leq Y. \]
\[ \text{min3 } @ \text{ min}(X,Y,Z) \land \text{ min}(X,Y,Z_1) \iff \text{ min}(X,Y,Z) \land Z=Z_1. \]

- Critical pairs of overlap between min1 and min3 joinable
- Critical pair from min2 and first head constraint of min3:
  \((\text{min}(X,Y,X) \land X=Z, X \leq Y \land \text{min}(X,Y,Z))\)
- Critical pair joined with rule r2:
  \(\text{r2 } @ X \leq Y \land \text{min}(X,Y,Z) \iff X \leq Y \land X=Z. \)
- Also joins c.p. from min2 and second head constraint of min3
Examples (VI)

Example (Minimum and partial order constraint (III))

\[
\begin{align*}
\text{min2 } & \quad @ \quad \min(X, Y, X) \iff X \leq Y. \\
\text{min3 } & \quad @ \quad \min(X, Y, Z) \land \min(X, Y, Z1) \iff \min(X, Y, Z) \land Z = Z1. \\
r2 & \quad @ \quad X \leq Y \land \min(X, Y, Z) \iff X \leq Y \land X = Z
\end{align*}
\]

- Critical pair of overlap between \text{min2} and \text{r2} joined by \text{r3} @ X \leq Y \land X \leq Y \iff X \leq Y.
- New rules \text{r1, r2, r3} reveal properties of \text{leq} and \text{min}
- Program with rules \text{min1, min2, min3} and \text{r1, r2, r3} is confluent and terminating
Examples (VII)

Example

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

$r_1 \@ p(X, Y) \iff X \geq Y \land q(X, Y)$.

$r_2 \@ p(X, Y) \iff X \leq Y \land r(X, Y)$.

- Critical pair from $r_1$ and $r_2$ not joinable

$$(X \geq Y \land q(X, Y), X \leq Y \land r(X, Y))$$

- Completion inserts two rules

$r_3 \@ r(X, Y) \iff X \leq Y \land q(X, Y) \land X \geq Y.$

$r_4 \@ q(X, Y) \Rightarrow X \geq Y \land X \leq Y.$
Examples (VIII)

Computations show that propagation rule $r_4$ is needed

**Example computation ($r_2$)**

- $p(X, Y)$
- $\mapsto \text{Apply } r_2 \quad r(X, Y) \land X \leq Y$
- $\mapsto \text{Apply } r_3 \quad q(X, Y) \land X = Y$
- $\mapsto \text{Apply } r_4 \quad q(X, Y) \land X = Y$

**Example computation ($r_1$)**

- $p(X, Y)$
- $\mapsto \text{Apply } r_1 \quad q(X, Y) \land X \geq Y$
- $\mapsto \text{Apply } r_4 \quad q(X, Y) \land X = Y$
Examples (IX)

Example

\[ r_1 \iff p(X, Y) \leftrightarrow X \geq Y \land q(X, Y). \]
\[ r_2 \iff p(X, Y) \leftrightarrow X \leq Y \land q(Y, X). \]

- Program not confluent, c.p. from \( r_1 \) and \( r_2 \) not joinable:

\[ (X \geq Y \land q(X, Y), \ X \leq Y \land q(Y, X)) \]

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

\[ r_f \iff q(Y, X) \leftrightarrow X \leq Y \land q(X, Y) \land X \geq Y. \]
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)

\[ p \iff q(X). \]
\[ p \iff \text{true}. \]

- Assume \( q \) does not hold for all possible values.
- Critical pair \((q(X), \text{true})\)
- Only way of orienting results in rule \( q(X) \Rightarrow \text{true} \)
- Contradicts logical reading since \( q \) does not hold for all values
Failing completion can exhibit inconsistency in program

Theorem

\( P \) a CHR program, \( CT \) 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 \( \{p \Leftrightarrow true, \ p \Leftrightarrow false\} \)
Examples (I)

Example

\( p(X) \iff q(X). \)

\( p(X) \iff \text{true}. \)

\( q(X) \iff X > 0. \)

- Critical pair \((X > 0, \text{true})\)
- 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)

\[ r_1 @ \text{max}(X, Y, Z) \iff X \leq Y \mid Z = Y. \]

\[ r_2 @ \text{max}(X, Y, Z) \iff Y \leq X \mid Y = X. \]

- Critical pair from $r_1$, $r_2$: not joinable and completion fails

\[(Z = Y \land X \leq Y \land Y \leq X, \ Y = X \land X \leq Y \land Y \leq X)\]
Example (Maximum with typo continued)

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

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

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

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 $P, CT \models false \leftrightarrow true$
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

\[ r_5 @ X \leq Y \iff X \neq Y \mid X < Y. \]

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

\[ r_6 @ X < Y \land Y < X \iff X \neq Y \mid false. \]

\[ r_7 @ X < Y \land X < Y \iff X \neq Y \mid X < Y. \]
Example (Defining < continued)

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

  \[(X = Y \land X \neq Y, \quad X < Y \land Y \leq X \land X \neq Y).\]

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

  \[(X \leq Y \land X \neq Y, \quad X < Y \land Y \leq X \land X \neq Y).\]

- **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)

\[ r_1 \equiv \text{member}(X, []) \equiv \text{false}. \]
\[ r_2 \equiv \text{member}(X, [H|T]) \equiv X = H \mid \text{true}. \]
\[ r_3 \equiv \text{member}(X, [H|T]) \equiv X \neq H \mid \text{member}(X, T). \]

- In CHR query \text{member}(X, [1, 2, 3]) delays
- Prolog computes answers \( X=1, X=2, X=3 \)
Example (Member predicate continued)

- **If rule added** $r_4$ **added, program loses confluence**

  $r_4 \ @ \ \text{member}(X, [1,2,3]) \iff \text{answer}(X),$

- **Completion generates rules equivalent to Prolog answers**

  $a_1 \ @ \ \text{answer}(1) \iff \text{true}.$
  $a_2 \ @ \ \text{answer}(2) \iff \text{true}.$
  $a_3 \ @ \ \text{answer}(3) \iff \text{true}.$
  $a_4 \ @ \ \text{answer}(X) \iff \neg X = 1 \land \neg X = 2 \land \neg X = 3 \mid \text{false}.$
Program specialization by completion (VI)

Example (Member predicate rule generation)

\[
\begin{align*}
\text{r4}@\text{member}(A, [1, 2, 3]) & \iff \text{answer}(A) \\
n1@\text{answer}(1) & \iff \text{true} & n2@\text{answer}(2) & \iff \text{true} & n3@\text{answer}(3) & \iff \text{true} \\
r5@\text{member}(A, [2, 3]) & \iff A \neq 1 \land \text{answer}(A) \\
r6@\text{member}(A, [3]) & \iff A \neq 1 \land A \neq 2 \land \text{answer}(A)
\end{align*}
\]
Program specialization by completion (VII)

Example

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

\[
\begin{align*}
X &= [a], \ Y = [c|Z] \\
X &= [a, b, c], \ Z = [b|Y] \\
X &= [a, b, c, X1], \ Z = [X1, b|Y] \\
X &= [a, b, c, X1, X2], \ Z = [X1, X2, b|Y] \\
&\ldots
\end{align*}
\]
Example

Applying completion to two rules of append and

\[ r3 \oplus \text{append}(X, [b|Y], [a,b,c|Z]) \iff \text{answer}(X, Y, Z). \]

results in

\[ a1 \oplus \text{answer}([a], [c|Z], Z) \iff \text{true}. \]
\[ a2 \oplus \text{answer}([a,b,c], Y, [b|Y]) \iff \text{true}. \]
\[ a3 \oplus \text{answer}([a,b,c,H|L], Y, [H|L2]) \]
\[ \iff \text{answer}([a,b,c|L], Y, L2). \]

- **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 \{a \leftrightarrow b\}, second program \{b \leftrightarrow a\}
- Both programs terminating
- Union \{a \leftrightarrow b\}, \{b \leftrightarrow a\} not terminating

Example
First program \{a \leftrightarrow b\}, second program \{a \leftrightarrow c\}
- Both programs confluent
- Union \{a \leftrightarrow b\}, \{a \leftrightarrow c\} is terminating but not confluent

⇒ Well-behavedness not preserved under union
Modular classes of CHR programs (I)

**Definition**

- **P, P₁, P₂** CHR programs
  - A **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 $P$)
  - $P₁, P₂$ **nonoverlapping** if they have no defined CHR constraints in common
  - $P₁, P₂$ **circular** if $P₁$ defines CHR constraint used in $P₂$ and vice versa
  - Given goal, variable $P₁, P₂$-**shared** if it occurs in constraints of $P₁$ and constraints of $P₂$
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

\[ P1: \quad c(f(X)) \iff X = g(Y) \land c(Y). \]
\[ P2: \quad d(g(Y)) \iff Y = f(Z) \land d(Z). \]

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

\[ \begin{align*}
X &= g(Y) \land c(Y) \land d(g(Y)) & \hookrightarrow P_1 \\
X &= g(f(W)) \land Y &= f(W) \land c(f(W)) \land d(W) & \hookrightarrow P_2 \ldots
\end{align*} \]

\[ \Rightarrow \text{Even noncircular definitions can lead to nontermination} \]
Examples (II)

Previous example, common function symbols replaced by built-ins

\[ P1: \quad c(FX) \Leftrightarrow f_1(FX, X) \land g_1(X, Y) \land c(Y). \]
\[ P2: \quad d(GY) \Leftrightarrow g_2(GY, Y) \land f_2(Y, Z) \land d(Z). \]

- \( f_1(X, Y) \) and \( f_2(X, Y) \) both defined as \( X = f(Y) \)
- \( g_1(X, Y) \) and \( g_2(X, Y) \) both defined as \( X = g(Y) \)
- \( c(FX) \land f_1(FX, X) \land d(X) \) 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

\( P1: \ c(X,N) \iff f(X,N) \land g(X,N) \land c(X,N+1). \)

\( P2: \ d(Y,N) \iff g(Y,N) \land f(Y,N+1) \land d(Y,N+1). \)

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

\( \Rightarrow \) Nontermination problem persists over \( P_1, P_2 \)-shared variables
Modularity of termination

- Common symbols influence 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)

$P_1$: \( c(X, N) \Leftrightarrow f(X, N) \mid g(X, N) \land c(X, N+1) \).

$P_2$: \( d(Y, N) \Leftrightarrow g(Y, N) \mid f(Y, N+1) \land d(Y, N+1) \).

- \( f(X, N) \) defined as \( X \mod 2^N = 0 \land X > N \land N > 0 \)
- \( g(X, N) \) defined as \( X \mod 3^N = 0 \land X > N \land N > 0 \)
- Programs noncircular, \( X \) and \( N \) \( P_1, P_2 \)-shared
- \( c(X, N) \land f(X, N) \land d(X, N) \) not terminating
Example (II)

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

Adding finite domain constraints in query

$X$ in $[2, 4, 6, 8]$ $\land$ $N$ in $[1, 2]$ $\land$ c($X$, $N$) $\land$ f($X$, $N$) $\land$ d($X$, $N$)

leads to state

$X$ in $[6]$ $\land$ $N$ in $[1]$ $\land$ g($X$, $N$) $\land$ f($X$, $N$) $\land$ d($X$, $N$) $\land$ 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**

\[ P_1, P_2 \text{ well-behaved and nonoverlapping, } P_1 \cup P_2 \text{ terminating}, \]

\[ \text{then } P_1 \cup P_2 \text{ is confluent} \]

**Example**

Union of well-behaved programs \{a ⇔ b\}, \{b ⇔ 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 \{a ⇔ b, a ⇔ c\} (both defining 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 $S \xrightarrow{+}_{P_{1}} S_{1}$ and $S \xrightarrow{+}_{P_{2}} S_{2}$ 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 (Two programs for \texttt{max})

\[
\begin{align*}
\text{max}(X, Y, Z) & \Leftrightarrow X \leq Y \mid Z = Y. & \text{max}(X, Y, Z) & \Leftrightarrow X < Y \mid Z = Y. \\
\text{max}(X, Y, Z) & \Leftrightarrow Y < X \mid Z = X. & \text{max}(X, Y, Z) & \Leftrightarrow Y \leq X \mid Z = X.
\end{align*}
\]

- \text{max}(X, Y, Z) \land X \leq Y \text{ 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 (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

Let $P_1$ be the program:

\[
p(a) \iff s.
\]

\[
p(b) \iff r.
\]

\[
s \land r \iff \text{true}.
\]

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 $P_1 p(a) \land p(b)$ leads to $\text{true}$, in $P_2$ to $s \land r$
- Including minimal states for $s$ and $r$ ($s \land r$) 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)**

The operation `sum(List, Sum)` holds if **Sum** is the sum of elements in **List**.

**Program \( P_1 \):**

\[
\begin{align*}
\text{sum}([], \text{Sum}) & \iff \text{Sum}=0. \\
\text{sum}([X|Xs], \text{Sum}) & \iff \text{sum}(Xs, \text{Sum1}) \land \text{Sum} = \text{Sum1}+X.
\end{align*}
\]

**Program \( P_2 \):**

\[
\begin{align*}
\text{sum}([], \text{Sum}) & \iff \text{Sum}=0. \\
\text{sum}([X|Xs], \text{Sum}) & \iff \text{sum1}(X, Xs, \text{Sum}). \\
\text{sum1}(X, [], \text{Sum}) & \iff \text{Sum}=X. \\
\text{sum1}(X, Xs, \text{Sum}) & \iff \text{sum}(Xs, \text{Sum1}) \land \text{Sum} = \text{Sum1}+X.
\end{align*}
\]
Examples (II)

**Example (Sum continued)**

sum-minimal states are \( \text{sum}([], \text{Sum}) \) and \( \text{sum}([X|Xs], \text{Sum}) \)

- For \( \text{sum}([], \text{Sum}) \) **final state** \( \text{Sum}=0 \) in \( P_1 \) and \( P_2 \)

- **Computation for** \( \text{sum}([X|Xs], \text{Sum}) \) **in** \( P_1 \):
  \[
  \text{sum}([X|Xs], \text{Sum}) \mapsto_{P_1} \text{sum}(Xs, \text{Sum}_1) \land \text{Sum} = \text{Sum}_1 + X
  \]

- **Computation for** \( \text{sum}([X|Xs], \text{Sum}) \) **in** \( P_2 \):
  \[
  \text{sum}([X|Xs], \text{Sum}) \mapsto_{P_2} \text{sum}_1(X, Xs, \text{Sum}) \mapsto_{P_2} \text{sum}(Xs, \text{Sum}_1) \land \text{Sum} = \text{Sum}_1 + X
  \]

All sum-minimal states \( P_1, P_2 \)-joinable

\( \Rightarrow \) \( P_1 \) and \( P_2 \) operationally sum-equivalent
Examples (III)

Example

Program $P_1$

\[ p(X) \Leftrightarrow X > 0 \mid q(X). \]
\[ q(X) \Leftrightarrow X < 0 \mid \text{true}. \]

Program $P_2$

\[ p(X) \Leftrightarrow X > 0 \mid q(X). \]
\[ q(X) \Leftrightarrow X < 0 \mid \text{false}. \]

$P_1, P_2$ operationally $p$-equivalent, but $p$-minimal state $q(X) \wedge X > 0$ 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 $P_1$

\begin{align*}
p & \iff s. \\
s \land q & \iff \text{true}.
\end{align*}

Program $P_2$

\begin{align*}
p & \iff s. \\
s \land q & \iff \text{false}.
\end{align*}

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

\[ P \setminus r \text{ denotes program } P \text{ without rule } r. \]

Rule \( r \) is redundant in \( P \) iff for all states \( S \)

\[ S \xrightarrow{\ast}_P S_1 \text{ then } S \xrightarrow{\ast}_{P \setminus r} S_2 \text{ such that } S_1 \equiv S_2 \]
Example (Union of \texttt{max} programs)

\begin{align*}
\texttt{r1} & @ \text{max}(X, Y, Z) \iff X < Y \mid Z = Y. \\
\texttt{r2} & @ \text{max}(X, Y, Z) \iff X \geq Y \mid Z = X. \\
\texttt{r3} & @ \text{max}(X, Y, Z) \iff X \leq Y \mid Z = Y. \\
\texttt{r4} & @ \text{max}(X, Y, Z) \iff X > Y \mid Z = X.
\end{align*}

\texttt{r3} can always be applied when \texttt{r1} can be applied with same answer (not vice versa) \Rightarrow \texttt{r1} is redundant, analogously \texttt{r4}
Theorem for redundant rules

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

**Theorem**

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

- Specialize equivalence test
  - Check if computation due to candidate rule for removal can be performed by \( P \setminus r \)
  - State in computation for minimal state of \( r \) must be reachable in \( P \setminus r \)
Examples (I)

Example (Union of max programs continued)

- \( r_1 \) @ \( \max(X, Y, Z) \iff X < Y \mid Z=Y \).
- \( r_2 \) @ \( \max(X, Y, Z) \iff X \geq Y \mid Z=X \).
- \( r_3 \) @ \( \max(X, Y, Z) \iff X \leq Y \mid Z=Y \).
- \( r_4 \) @ \( \max(X, Y, Z) \iff X > Y \mid Z=X \).

- Any subset of program still well-behaved
- Removal of rule \( r_1 \) (min. state \( \max(X, Y, Z) \land X < Y \)), run
  
  \[ P : \ max(X, Y, Z) \land X < Y \rightarrow X < Y \land Z = Y \] by rule \( r_1 \)

  \[ P \setminus \{ r_1 \} : \ max(X, Y, Z) \land X < Y \rightarrow X < Y \land Z = Y \] by rule \( r_3 \)

- \( r_3 \) enables same computation \( \Rightarrow \) \( r_1 \) redundant
- Redundancy of \( r_4 \) shown analogously
Examples (II)

Example (Union of $\text{max}$ programs continued)

- $r_1 \ @ \ \text{max}(X, Y, Z) \iff X < Y \mid Z = Y.$
- $r_2 \ @ \ \text{max}(X, Y, Z) \iff X \geq Y \mid Z = X.$
- $r_3 \ @ \ \text{max}(X, Y, Z) \iff X \leq Y \mid Z = Y.$
- $r_4 \ @ \ \text{max}(X, Y, Z) \iff X > Y \mid Z = X.$

- **Rule $r_2$ not redundant**

  $P: \ \text{max}(X, Y, Z) \land X \geq Y \iff X \geq Y \land Z = X$  
  by rule $r_2$

  $P \setminus \{r_2\}: \ \text{max}(X, Y, Z) \land X \geq Y \not\iff$

- **Program without redundant rules consists of $r_2$ and $r_3$**
Examples (III)

Example (Strict order relation)

duplicate @ X less Y \ X less Y \equiv true.

irreflexivity @ X less X \equiv false.

antisymmetry @ X less Y \land Y less X \equiv false.

transitivity @ X less Y \land Y less Z \Rightarrow X less Z.

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

Resulting program not necessarily unique

Example

\[ r_1 @ p \iff q. \]
\[ r_2 @ p \iff \text{false}. \]
\[ r_3 @ q \iff \text{false}. \]

- Either \( r_1 \) or \( r_2 \) 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
    - 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 $\text{even}$ constraint)

$\text{even}(s(N)) \iff N = s(M) \land \text{even}(M)$.

- Time complexity of single $\text{even}$ linear in derivation length (rank)
- Time complexity of several ground $\text{even}$ constraints also linear (where rank is sum of ranks of individual constraints)
Example (II) (Complexity of even constraint continued)

- Adding second rule
  
  \[\text{even}(s(N)) \iff N=s(M) \land \text{even}(M).\]
  
  \[\text{even}(s(X)) \land \text{even}(X) \iff \text{false} .\]

- 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**

\[ r \text{ a simplification rule } H \Leftrightarrow G \setminus C \land B \]

\((H \text{ conjunction of } n \text{ CHR constraints, } C, B \text{ built-ins, } B \text{ CHR constraints})\)

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**

If $P$ contains 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

$$O\left(\sum_i (D^{n_i+1} O_{G_i} + DO_{C_i})\right)$$
Example (One rule program with successor notation)

\[ c(s(X)) \Leftrightarrow c(X) \land c(X). \]

- Removing successor doubles number of constraints
- Exponential ranking needed
  - \( \text{rank}(c(t)) = 2^{\text{size}(t)} - 1 \)
  - \( \text{size}(0) = 0 \)
  - \( \text{size}(s(N)) = 1 + \text{size}(N) \)
- Complexity exponential in size of argument of \( c(n = \text{size}(t)) \):
  \[
  O(2^n ((1 + 2^n)^1 (1 + 0) + (0 + 1))) = O(2^n 2^n) = O(4^n)
  \]
- \( O(2^n) \) derivation steps, \( O(2^n) \) constraints in each state