# COMP4011/8011
# Advanced Topics in Formal Methods and Programming Languages

## – Software Verification with Isabelle/HOL –

Peter Höfner

August 18, 2024

# Section 9

## Term Rewriting

# The Problem

Given a set of equations

$$\begin{gathered} l_1 = r_1 \\ l_2 = r_2 \\ \vdots \\ l_n = r_n \end{gathered}$$

does equation $l = r$ hold?

Applications in:

- Mathematics (algebra, group theory, etc)
- Functional Programming (model of execution)
- Theorem Proving (dealing with equations, simplifying statements)

# Term Rewriting: The Idea

use equations as reduction rules

$$\begin{gathered} l_1 \longrightarrow r_1 \\ l_2 \longrightarrow r_2 \\ \vdots \\ l_n \longrightarrow r_n \end{gathered}$$

decide $l = r$ by deciding $l \overset{*}{\longleftrightarrow} r$

# Arrow Cheat Sheet

| | | | |
|---|---|---|---|
| $\xrightarrow{0}$ | $=$ | $\{(x,y) \mid x = y\}$ | identity |
| $\xrightarrow{n+1}$ | $=$ | $\xrightarrow{n} \circ \longrightarrow$ | n+1 fold composition |
| $\xrightarrow{+}$ | $=$ | $\bigcup_{i>0} \xrightarrow{i}$ | transitive closure |
| $\xrightarrow{*}$ | $=$ | $\xrightarrow{+} \cup \xrightarrow{0}$ | reflexive transitive closure |
| $\xrightarrow{=}$ | $=$ | $\longrightarrow \cup \xrightarrow{0}$ | reflexive closure |
| $\xrightarrow{-1}$ | $=$ | $\{(y,x) \mid x \longrightarrow y\}$ | inverse |
| $\longleftarrow$ | $=$ | $\xrightarrow{-1}$ | inverse |
| $\longleftrightarrow$ | $=$ | $\longleftarrow \cup \longrightarrow$ | symmetric closure |
| $\xleftrightarrow{+}$ | $=$ | $\bigcup_{i>0} \xleftrightarrow{i}$ | transitive symmetric closure |
| $\xleftrightarrow{*}$ | $=$ | $\xleftrightarrow{+} \cup \xleftrightarrow{0}$ | reflexive transitive symmetric closure |

# How to Decide $l \overset{*}{\longleftrightarrow} r$

**Same idea as for $\beta$:** look for $n$ such that $l \xrightarrow{*} n$ and $r \xrightarrow{*} n$

Does this always work?

If $l \xrightarrow{*} n$ and $r \xrightarrow{*} n$ then $l \overset{*}{\longleftrightarrow} r$. Ok.
If $l \overset{*}{\longleftrightarrow} r$, will there always be a suitable $n$? **No!**

Example:

Rules: $f\ x \longrightarrow a, \quad g\ x \longrightarrow b, \quad f\ (g\ x) \longrightarrow b$
$f\ x \overset{*}{\longleftrightarrow} g\ x$ because $f\ x \longrightarrow a \longleftarrow f\ (g\ x) \longrightarrow b \longleftarrow g\ x$
**But:** $f\ x \longrightarrow a$ and $g\ x \longrightarrow b$ and $a$, $b$ in normal form

Works only for systems with **Church-Rosser** property:

$$l \overset{*}{\longleftrightarrow} r \Longrightarrow \exists n.\ l \xrightarrow{*} n \wedge r \xrightarrow{*} n$$

**Fact:** $\longrightarrow$ is Church-Rosser iff it is confluent.

# Confluence

$$\begin{tikzcd}
& s \arrow[dl, "*"'] \arrow[dr, "*"] & \\
x \arrow[dr, dashed, "*"'] & & y \arrow[dl, dashed, "*"] \\
& t &
\end{tikzcd}$$

**Problem:**
is a given set of reduction rules confluent?

**undecidable**

## Local Confluence

$$\begin{tikzcd}
& s \arrow[dl] \arrow[dr] & \\
x \arrow[dr, dashed, "*"'] & & y \arrow[dl, dashed, "*"] \\
& t &
\end{tikzcd}$$

**Fact:** local confluence and termination $\Longrightarrow$ confluence

# Termination

$\longrightarrow$ is **terminating** if there are no infinite reduction chains
$\longrightarrow$ is **normalizing** if each element has a normal form
$\longrightarrow$ is **convergent** if it is terminating and confluent

Example:
$\longrightarrow_\beta$ in $\lambda$ is not terminating, but confluent
$\longrightarrow_\beta$ in $\lambda^\rightarrow$ is terminating and confluent, i.e. convergent

**Problem:** is a given set of reduction rules terminating?

**undecidable**

# When is $\longrightarrow$ Terminating?

**Basic idea:** when each rule application makes terms simpler in some way.

More formally: $\longrightarrow$ is terminating when there is a well founded
order $<$ on terms for which $s < t$ whenever $t \longrightarrow s$
(well founded = no infinite decreasing chains $a_1 > a_2 > \ldots$)

Example: $f\ (g\ x) \longrightarrow g\ x$, $g\ (f\ x) \longrightarrow f\ x$

This system always terminates. Reduction order:

$s <_r t$ iff $\mathit{size}(s) < \mathit{size}(t)$ with
$\mathit{size}(s) =$ number of function symbols in $s$

1. Both rules always decrease *size* by 1 when applied to any term $t$
2. $<_r$ is well founded, because $<$ is well founded on $\mathbb{N}$

# Termination in Practice

**In practice:** often easier to consider just the rewrite rules by themselves, rather than their application to an arbitrary term $t$.

**Show** for each rule $l_i = r_i$, that $r_i < l_i$.

Example:

$g\ x < f\ (g\ x)$ and $f\ x < g\ (f\ x)$

Requires

$u$ to become smaller whenever any subterm of $u$ is made smaller.

**Formally:**

Requires $<$ to be **monotonic** with respect to the structure of terms:

$$s < t \longrightarrow u[s] < u[t].$$

True for most orders that don't treat certain parts of terms as special cases.

# Example Termination Proof

**Problem:** Rewrite formulae containing $\neg$, $\wedge$, $\vee$ and $\longrightarrow$, so that they don't contain any implications and $\neg$ is applied only to variables and constants.

**Rewrite Rules:**

- Remove implications:

  **imp:** $(A \longrightarrow B) = (\neg A \vee B)$

- Push $\neg$s down past other operators:

  **notnot:** $(\neg\neg P) = P$

  **notand:** $(\neg(A \wedge B)) = (\neg A \vee \neg B)$

  **notor:** $(\neg(A \vee B)) = (\neg A \wedge \neg B)$

We show that the rewrite system defined by these rules is terminating.

# Order on Terms

Each time one of our rules is applied, either:

- an implication is removed, or
- something that is not a $\neg$ is hoisted upwards in the term.

This suggests a 2-part order, $<_r$: $s <_r t$ iff:

- $\mathsf{num\_imps}\ s < \mathsf{num\_imps}\ t$, or
- $\mathsf{num\_imps}\ s = \mathsf{num\_imps}\ t \wedge \mathsf{osize}\ s < \mathsf{osize}\ t$.

Let:

- $s <_i t \equiv \mathsf{num\_imps}\ s < \mathsf{num\_imps}\ t$ and
- $s <_n t \equiv \mathsf{osize}\ s < \mathsf{osize}\ t$

Then $<_i$ and $<_n$ are both well-founded orders (since both return nats). $<_r$ is the lexicographic order over $<_i$ and $<_n$. $<_r$ is well-founded since $<_i$ and $<_n$ are both well-founded.

# Order Decreasing

**imp** clearly decreases num_imps.
osize adds up all non-$\neg$ operators and variables/constants, weights each one according to its depth within the term.

$$\begin{array}{lll}
\mathrm{osize}'\ c & x & = 2^x \\
\mathrm{osize}'\ (\neg P) & x & = \mathrm{osize}'\ P\ (x+1) \\
\mathrm{osize}'\ (P \wedge Q) & x & = 2^x + (\mathrm{osize}'\ P\ (x+1)) + (\mathrm{osize}'\ Q\ (x+1)) \\
\mathrm{osize}'\ (P \vee Q) & x & = 2^x + (\mathrm{osize}'\ P\ (x+1)) + (\mathrm{osize}'\ Q\ (x+1)) \\
\mathrm{osize}'\ (P \longrightarrow Q) & x & = 2^x + (\mathrm{osize}'\ P\ (x+1)) + (\mathrm{osize}'\ Q\ (x+1)) \\
\mathrm{osize}\ \ P & & = \mathrm{osize}'\ P\ 0
\end{array}$$

The other rules decrease the depth of the things osize counts, so decrease osize.

# Term Rewriting in Isabelle

Term rewriting engine in Isabelle is called **Simplifier**

**apply** simp

- uses simplification rules
- (almost) blindly from left to right
- until no rule is applicable.

| | |
|---|---|
| termination: | not guaranteed<br>(may loop) |
| confluence: | not guaranteed<br>(result may depend on which rule is used first) |

# Control

- Equations turned into simplification rules with **[simp]** attribute
- Adding/deleting equations locally:

  **apply** (simp add: <rules>)    and    **apply** (simp del: <rules>)
- Using only the specified set of equations:

  **apply** (simp only: <rules>)

# Demo

# Exercises

- Show, via a pen-and-paper proof, that the osize function is monotonic with respect to the structure of terms from that example.

# Applying a Rewrite Rule

- $l \longrightarrow r$ applicable to term $t[s]$
  if there is substitution $\sigma$ such that $\sigma\ l = s$
- Result: $t[\sigma\ r]$
- Equationally: $t[s] = t[\sigma\ r]$

Example:

**Rule:** $0 + n \longrightarrow n$

**Term:** $a + (0 + (b + c))$

**Substitution:** $\sigma = \{n \mapsto b + c\}$

**Result:** $a + (b + c)$

# Conditional Term Rewriting

Rewrite rules can be conditional:

$$\llbracket P_1 \dots P_n \rrbracket \Longrightarrow l = r$$

is applicable to term $t[s]$ with $\sigma$ if

- $\sigma\ l = s$ and
- $\sigma\ P_1, \dots, \sigma\ P_n$ are provable by rewriting.

# Rewriting with Assumptions

Isabelle uses assumptions in rewriting.

**Can lead to non-termination.**

Example:

**lemma** ”$f\ x = g\ x \wedge g\ x = f\ x \Longrightarrow f\ x = 2$¨

| | |
|---|---|
| simp | **use and simplify** assumptions |
| (simp (no_asm)) | **ignore** assumptions |
| (simp (no_asm_use)) | **simplify**, but do **not use** assumptions |
| (simp (no_asm_simp)) | **use**, but do **not simplify** assumptions |

# Preprocessing

Preprocessing (recursive) for maximal simplification power:

$$\begin{array}{rcl}
\neg A & \mapsto & A = \mathit{False} \\
A \longrightarrow B & \mapsto & A \Longrightarrow B \\
A \wedge B & \mapsto & A,\ B \\
\forall x.\ A\ x & \mapsto & A\ ?x \\
A & \mapsto & A = \mathit{True}
\end{array}$$

Example: $$(p \longrightarrow q \wedge \neg r) \wedge s$$

$$\mapsto$$

$$p \Longrightarrow q = \mathit{True} \qquad p \Longrightarrow r = \mathit{False} \qquad s = \mathit{True}$$

# Demo

# Case splitting with simp

$P\ (\text{if } A \text{ then } s \text{ else } t) \ = \ (A \longrightarrow P\ s) \wedge (\neg A \longrightarrow P\ t)$

Automatic

$P\ (\text{case } e \text{ of } 0 \ \Rightarrow\ a \mid \text{Suc } n \ \Rightarrow\ b) \ =$
$\quad (e = 0 \longrightarrow P\ a) \wedge (\forall n.\ e = \text{Suc } n \longrightarrow P\ b)$

Manually: **apply** (simp split: nat.split)

Similar for any data type t: **t.split**

# Congruence Rules

**congruence rules are about using context**

Example: in $P \longrightarrow Q$ we could use $P$ to simplify terms in $Q$

For $\Longrightarrow$ hardwired (assumptions used in rewriting)

For other operators expressed with conditional rewriting.

Example: $[\![P = P'; P' \Longrightarrow Q = Q']\!] \Longrightarrow (P \longrightarrow Q) = (P' \longrightarrow Q')$

Read: to simplify $P \longrightarrow Q$

- first simplify $P$ to $P'$
- then simplify $Q$ to $Q'$ using $P'$ as assumption
- the result is $P' \longrightarrow Q'$

# More Congruence

Sometimes useful, but not used automatically (slowdown):

**conj_cong**: $\llbracket P = P'; P' \Longrightarrow Q = Q' \rrbracket \Longrightarrow (P \wedge Q) = (P' \wedge Q')$

Context for if-then-else:

**if_cong**: $\llbracket b = c; c \Longrightarrow x = u; \neg c \Longrightarrow y = v \rrbracket \Longrightarrow$
$(\text{if } b \text{ then } x \text{ else } y) = (\text{if } c \text{ then } u \text{ else } v)$

Prevent rewriting inside then-else (default):

**if_weak_cong**: $b = c \Longrightarrow (\text{if } b \text{ then } x \text{ else } y) = (\text{if } c \text{ then } x \text{ else } y)$

- declare own congruence rules with **[cong]** attribute
- delete with **[cong del]**
- use locally with e.g. **apply** (simp cong: <rule>)

# Ordered rewriting

**Problem:** $x + y \longrightarrow y + x$ does not terminate

**Solution:** use permutative rules only if term becomes lexicographically smaller.

Example: $b + a \rightsquigarrow a + b$ but not $a + b \rightsquigarrow b + a$.

For types nat, int etc:

- lemmas **add_ac** sort any sum $(+)$
- lemmas **mult_ac** sort any product $(*)$

Example: **apply** (simp add: add_ac) yields
$(b + c) + a \rightsquigarrow \cdots \rightsquigarrow a + (b + c)$

# AC Rules

**Example for associative-commutative rules:**

Associative: $(x \odot y) \odot z = x \odot (y \odot z)$

Commutative: $x \odot y = y \odot x$

These 2 rules alone get stuck too early (not confluent).

Example: $(z \odot x) \odot (y \odot v)$

We want: $(z \odot x) \odot (y \odot v) = v \odot (x \odot (y \odot z))$

We get: $(z \odot x) \odot (y \odot v) = v \odot (y \odot (x \odot z))$

**We need:** **AC rule** $x \odot (y \odot z) = y \odot (x \odot z)$

If these 3 rules are present for an AC operator
Isabelle will order terms correctly

# Demo

# Back to Confluence

**Remember:** confluence in general is undecidable.
But: confluence for terminating systems is decidable!
Problem: overlapping lhs of rules.

**Definition:**

Let $l_1 \longrightarrow r_1$ and $l_2 \longrightarrow r_2$ be two rules with disjoint variables.
They form a **critical pair** if a non-variable subterm of $l_1$ unifies with $l_2$.

Example:
Rules: (1) $f\ x \longrightarrow a$ (2) $g\ y \longrightarrow b$ (3) $f\ (g\ z) \longrightarrow b$
Critical pairs:

(1)+(3) $\{x \mapsto g\ z\}$ $a \xleftarrow{(1)} f\ (g\ z) \xrightarrow{(3)} b$

(3)+(2) $\{z \mapsto y\}$ $b \xleftarrow{(3)} f\ (g\ y) \xrightarrow{(2)} f\ b$

# Completion

$$(1)\ f\ x \longrightarrow a \qquad (2)\ g\ y \longrightarrow b \qquad (3)\ f\ (g\ z) \longrightarrow b$$

is not confluent

But it can be made confluent by adding rules!

How: join all critical pairs

Example:

$$(1){+}(3) \qquad \{x \mapsto g\ z\} \qquad a \xleftarrow{(1)} f\ (g\ z) \xrightarrow{(3)} b$$

shows that $a = b$ (because $a \overset{*}{\longleftrightarrow} b$), so we add $a \longrightarrow b$ as a rule

This is the main idea of the Knuth-Bendix completion algorithm.

# Orthogonal Rewriting Systems

**Definitions:**

A **rule** $l \longrightarrow r$ is **left-linear** if no variable occurs twice in $l$.
A **rewrite system** is **left-linear** if all rules are.

A system is **orthogonal** if it is left-linear and has no critical pairs.

**Orthogonal rewrite systems are confluent**

Application: functional programming languages

# We have learned ...

- Conditional term rewriting
- Congruence rules
- AC rules
- More on confluence