Content

→ Intro & motivation, getting started

→ Foundations & Principles
  • Lambda Calculus, natural deduction
    [1, 2]
  • Higher Order Logic
    [3]
  • Term rewriting
    [4]

→ Proof & Specification Techniques
  • Inductively defined sets, rule induction
    [5]
  • Datatypes, recursion, induction
    [6, 7]
  • Hoare logic, proofs about programs, C verification
    [8, 9]
  • (mid-semester break)
  • Writing Automated Proof Methods
    [10]
  • Isar, codegen, typeclasses, locales
    [11, 12]

\[a1 \text{ due; } b2 \text{ due; } c3 \text{ due}\]
Last Time on HOL

- Defining HOL
- Higher Order Abstract Syntax
- Deriving proof rules
- More automation
Term Rewriting
The Problem

Given a set of equations

\[ l_1 = r_1 \]
\[ l_2 = r_2 \]
\[ \vdots \]
\[ l_n = r_n \]

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

\[ l_1 \rightarrow r_1 \]
\[ l_2 \rightarrow r_2 \]
\[ \vdots \]
\[ l_n \rightarrow r_n \]

decide \( l = r \) by deciding \( l \leftrightarrow^* r \)
Arrow Cheat Sheet

\[ \begin{align*}
0 & \rightarrow \quad = \quad \{(x, y) | x = y\} \quad \text{identity} \\
(n + 1) & \rightarrow \quad = \quad n \rightarrow \circ \rightarrow \quad \text{n+1 fold composition} \\
(+) & \rightarrow \quad = \quad \bigcup_{i>0} i \rightarrow \quad \text{transitive closure} \\
(*) & \rightarrow \quad = \quad \bigcup \rightarrow \quad \text{reflexive transitive closure} \\
(=) & \rightarrow \quad = \quad \bigcup \rightarrow \quad \text{reflexive closure} \\
(-1) & \rightarrow \quad = \quad \{(y, x) | x \rightarrow y\} \quad \text{inverse} \\
\leftrightarrow & \quad = \quad -1 \rightarrow \quad \text{inverse} \\
\leftrightarrow & \quad = \quad \leftrightarrow \bigcup \rightarrow \quad \text{symmetric closure} \\
(+\leftrightarrow) & \quad = \quad \bigcup_{i>0} i \leftrightarrow \quad \text{transitive symmetric closure} \\
(*\leftrightarrow) & \quad = \quad \bigcup \leftrightarrow \bigcup \leftrightarrow \quad \text{reflexive transitive symmetric closure}
\end{align*} \]
How to Decide \( l \xleftrightarrow{\ast} r \)

Same idea as for \( \beta \): look for \( n \) such that \( l \rightarrow^* n \) and \( r \rightarrow^* n \)

Does this always work?  
If \( l \rightarrow^* n \) and \( r \rightarrow^* n \) then \( l \xleftrightarrow{\ast} r \). Ok.

If \( l \xleftrightarrow{\ast} r \), will there always be a suitable \( n \)? No!

Example:
Rules:  
\[
\begin{align*}
  f \; x &\rightarrow a, \quad g \; x \rightarrow b, \quad f \; (g \; x) &\rightarrow b \\
  f \; x &\xleftrightarrow{\ast} g \; x \quad \text{because} \quad f \; x \rightarrow a \leftarrow f \; (g \; x) \rightarrow b \leftarrow g \; x
\end{align*}
\]

But: \( f \; x \rightarrow a \) and \( g \; x \rightarrow b \) and \( a, b \) in normal form

Works only for systems with Church-Rosser property:
\[
l \xleftrightarrow{\ast} r \iff \exists n. \; l \rightarrow^* n \land r \rightarrow^* n
\]

Fact: \( \rightarrow \) is Church-Rosser iff it is confluent.
Confluence

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

undecidable

Local Confluence

**Fact:** local confluence and termination $\implies$ confluence
Termination

\[ \rightarrow \] is **terminating** if there are no infinite reduction chains

\[ \rightarrow \] is **normalizing** if each element has a normal form

\[ \rightarrow \] is **convergent** if it is terminating and confluent

**Example:**

\[ \rightarrow^\beta \] in \( \lambda \) is not terminating, but confluent

\[ \rightarrow^\beta \] 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 \( t \longrightarrow s \)

\( \text{(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:

\[ s <_r t \iff \text{size}(s) < \text{size}(t) \]

\[ \text{size}(s) = \text{number of function symbols in } s \]

1. Both rules always decrease \( \text{size} \) 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:

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

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:

$s < t \implies u[s] < u[t]$. True for most orders that don’t treat certain parts of terms as special cases.
Example Termination Proof

**Problem:** Rewrite formulae containing $\neg$, $\land$, $\lor$ and $\rightarrow$, so that they don’t contain any implications and $\neg$ is applied only to variables and constants.

**Rewrite Rules:**

- **Remove implications:**
  
  $\textbf{imp: } (A \rightarrow B) = (\neg A \lor B)$

- **Push $\neg$s down past other operators:**
  
  $\textbf{notnot: } (\neg\neg P) = P$
  
  $\textbf{notand: } (\neg(A \land B)) = (\neg A \lor \neg B)$
  
  $\textbf{notor: } (\neg(A \lor B)) = (\neg A \land \neg B)$

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 ¬ is hoisted upwards in the term.

This suggests a 2-part order, \(<_r\): s \(<_r\) t iff:

→ num_imps s \(<\) num_imps t, or
→ num_imps s = num_imps t \(\land\) osize s \(<\) osize t.

Let:

→ s \(<_i\) t \(\equiv\) num_imps s \(<\) num_imps t and
→ s \(<_n\) t \(\equiv\) osize s \(<\) 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.

\[
\begin{align*}
o_{size}' \ c & \quad x = 2^x \\
o_{size}' \ (\neg P) & \quad x = o_{size}' P \ (x + 1) \\
o_{size}' \ (P \land Q) & \quad x = 2^x + (o_{size}' P \ (x + 1)) + (o_{size}' Q \ (x + 1)) \\
o_{size}' \ (P \lor Q) & \quad x = 2^x + (o_{size}' P \ (x + 1)) + (o_{size}' Q \ (x + 1)) \\
o_{size}' \ (P \rightarrow Q) & \quad x = 2^x + (o_{size}' P \ (x + 1)) + (o_{size}' Q \ (x + 1)) \\
o_{size} P & \quad = o_{size}' P 0
\end{align*}
\]

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
(may loop)

**confluence:** not guaranteed
(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
We have seen today...

- Equations and Term Rewriting
- Confluence and Termination of reduction systems
- Term Rewriting in Isabelle
Exercises

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