COMP 4161
NICTA Advanced Course

Advanced Topics in Software Verification

Gerwin Klein, June Andronick, Toby Murray
Content

➜ Intro & motivation, getting started

➜ Foundations & Principles
  • Lambda Calculus, natural deduction [2,3,4\textsuperscript{a}]
  • Higher Order Logic [5,6\textsuperscript{b},7]
  • Term rewriting [8,9,10\textsuperscript{c}]

➜ Proof & Specification Techniques
  • Isar [11,12\textsuperscript{d}]
  • Inductively defined sets, rule induction [13\textsuperscript{e},15]
  • Datatypes, recursion, induction [16,17\textsuperscript{f},18,19]
  • Calculational reasoning, mathematics style proofs [20]
  • Hoare logic, proofs about programs [21\textsuperscript{g},22,23]

\textsuperscript{a} a1 out; \textsuperscript{b} a1 due; \textsuperscript{c} a2 out; \textsuperscript{d} a2 due; \textsuperscript{e} session break; \textsuperscript{f} a3 out; \textsuperscript{g} a3 due
Last Time on HOL

► Defining HOL
► Higher Order Abstract Syntax
► Deriving proof rules
► More automation
The Three Basic Ways of Introducing Theorems

➜ Axioms:

Example: \texttt{axioms refl: }"t = t"

Do not use. Evil. Can make your logic inconsistent.

➜ Definitions:

Example: \texttt{definition inj where }"inj f \equiv \forall x y. f x = f y \rightarrow x = y"

Introduces a new lemma called inj_def.

➜ Proofs:

Example: \texttt{lemma }"inj (\lambda x. x + 1)"

The harder, but safe choice.
The Three Basic Ways of Introducing Types

- **typedecl**: by name only
  
  Example: `typedecl names`
  
  Introduces new type `names` without any further assumptions

- **types**: by abbreviation
  
  Example: `types α rel = "α ⇒ α ⇒ bool"`
  
  Introduces abbreviation `rel` for existing type `α ⇒ α ⇒ bool`
  
  Type abbreviations are immediately expanded internally

- **typedef**: by definition as a set
  
  Example: `typedef new_type = "{some set}" <proof>`
  
  Introduces a new type as a subset of an existing type.
  The proof shows that the set on the rhs in non-empty.
  More on `typedef` in later lectures.
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)
use equations as reduction rules

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

deceive \( l = r \) by deciding \( l \leftarrow^* r \)
Arrow Cheat Sheet

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

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 $l \leftrightarrow^* r$. Ok.

If $l \leftrightarrow^* r$, will there always be a suitable $n$? No!

Example:

Rules:

\[
\begin{align*}
    f \; x & \rightarrow a, \quad g \; x & \rightarrow b, \quad f \; (g \; x) & \rightarrow b \\
    f \; x & \leftrightarrow^* g \; x \quad \text{because} \quad f \; x & \rightarrow a \leftarrow f \; (g \; x) & \rightarrow b \leftarrow g \; x \\
    \text{But:} \quad f \; x & \rightarrow a \quad \text{and} \quad g \; x & \rightarrow b \quad \text{and} \quad a, b \text{ in normal form}
\end{align*}
\]

Works only for systems with Church-Rosser property:

\[
l \leftrightarrow^* r \Rightarrow \exists n. \; l \rightarrow^* n \land r \rightarrow^* n
\]

Fact: $\rightarrow$ is Church-Rosser iff it is confluent.
Problem:
is a given set of reduction rules confluent?

undecidable

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

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

**Example:**

→_β_ in λ is not terminating, but confluent
→_β_ in λ → is terminating and confluent, i.e. convergent

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

**undecidable**
When is \( \rightarrow \) Terminating?

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

**More formally:** \( \rightarrow \) is terminating when

there is a well founded order \(<\) on terms for which \( s < t \) whenever \( t \rightarrow s \)

(Well founded = no infinite decreasing chains \( a_1 > a_2 > \ldots \))

**Example:** \( f (g x) \rightarrow g x, g (f x) \rightarrow f x \)

This system always terminates. Reduction order:

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

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

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

$$g \, x <_r \, f \, (g \, x) \text{ and } f \, x < g \, (f \, x)$$

Requires $t$ to become smaller whenever any subterm of $t$ 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:**

$\rightarrow$ Remove implications:

$\text{imp: } (A \rightarrow B) = (\neg A \lor B)$

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

$\text{notnot: } (\neg \neg P) = P$

$\text{notand: } (\neg (A \land B)) = (\neg A \lor \neg B)$

$\text{notor: } (\neg (A \lor B)) = (\neg A \land \neg B)$

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 \( \neg \) is hoisted upwards in the term.

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

- \( \text{num}_\text{imps} \ s < \text{num}_\text{imps} \ t \), or
- \( \text{num}_\text{imps} \ s = \text{num}_\text{imps} \ t \land \text{osize} \ s < \text{osize} \ t \).

Let:

- \( s <_i t \equiv \text{num}_\text{imps} \ s < \text{num}_\text{imps} \ t \) and
- \( s <_n t \equiv \text{osize} \ s < \text{osize} \ t \)

Then \( <_i \) and \( <_n \) are both well-founded orders (since both functions 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.

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

The other rules decrease the depth of the things **osize** counts, so decrease **osize**.
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)
Equations turned into simplification rules with \texttt{[simp]} attribute

Adding/deleting equations locally:
\begin{itemize}
    \item \texttt{apply} (simp add: \texttt{<rules>}) and \texttt{apply} (simp del: \texttt{<rules>})
\end{itemize}

Using only the specified set of equations:
\begin{itemize}
    \item \texttt{apply} (simp only: \texttt{<rules>})
\end{itemize}
DEMO
We have seen today...

- Equations and Term Rewriting
- Confluence and Termination of reduction systems
- Term Rewriting in Isabelle
Show, via a pen-and-paper proof, that the osize function is monotonic with respect to the structure of terms from that example.