COMP 4161
NICTA Advanced Course

Advanced Topics in Software Verification

Simon Winwood, Toby Murray, June Andronick, Gerwin Klein

Slide 1

Content

• Intro & motivation, getting started with Isabelle
• Foundations & Principles
  • Lambda Calculus
  • Higher Order Logic, natural deduction
  • Term rewriting
• Proof & Specification Techniques
  • Inductively defined sets, rule induction
  • Datatypes, recursion, induction
  • Calculational reasoning, mathematics style proofs
  • Hoare logic, proofs about programs

Slide 2

Last Time on HOL

• Defining HOL
• Higher Order Abstract Syntax
• Deriving proof rules
• More automation

Slide 3

The Three Basic Ways of Introducing Theorems

• Axioms:
  Example: \texttt{axioms refl: \texttt{"1 = 1"}}
  Do not use. Evil. Can make your logic inconsistent.

• Definitions:
  Example: \texttt{\texttt{defs inj_def: \texttt{"inj f \equiv \forall x y. f x = f y \rightarrow x = y"}}}

• Proofs:
  Example: \texttt{\texttt{lemma \texttt{"inj (\lambda x. x + 1)"}}}
  The harder, but safe choice.

Slide 4
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 is non-empty.

---

Example: Pairs

1. Pick existing type: `α ⇒ β ⇒ bool`
2. Identify subset:
   
   \[(α, β) \text{ Prod} = \{ f | \exists a b. f = \lambda(x :: α)(y :: β). x = a \land y = b \}\]
3. We get from Isabelle:
   
   - functions `AbsProd`, `RepProd`
   - both injective
   - `AbsProd (RepProd x) = x`
4. We now can:
   
   - define constants `Pair`, `fst`, `snd` in terms of `AbsProd` and `RepProd`
   - derive all characteristic theorems
   - forget about `Rep/Abs`, use characteristic theorems instead
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 \( l \overset{*}{\rightarrow} r \)
Arrow Cheat Sheet

- $n \equiv (x, y) \ | x = y$ identity
- $(x, y) \ | x = y$ identity
- $n + 1 \Rightarrow n$ fold composition
- $\Rightarrow = \bigcup_{i>0} i$ transitive closure
- $\Rightarrow = \bigcup \bigcup 0$ reflexive transitive closure
- $\Rightarrow = \bigcup \bigcup 0$ reflexive closure
- $(y, x) \ | x \Rightarrow y$ inverse
- $\Leftrightarrow = \bigcup_{i>0} i$ transitive symmetric closure
- $\Leftrightarrow = \bigcup \bigcup 0$ reflexive transitive symmetric closure

How to Decide $l \Rightarrow r$

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 $l \Rightarrow r$. Ok.
- If $l \Rightarrow r$, will there always be a suitable $n$? No!

Example:
Rules: $f x \Rightarrow a$, $g z \Rightarrow b$, $f (g x) \Rightarrow b$
$f x \Rightarrow g x$ because $f x \Rightarrow a \Rightarrow f (g x) \Rightarrow b \Rightarrow g x$
But: $f x \Rightarrow a$ and $g x \Rightarrow b$ and $a, b$ in normal form

Works only for systems with Church-Rosser property:
- $l \Rightarrow r \Rightarrow \exists n. l \Rightarrow n \land r \Rightarrow n$

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 $\Rightarrow$ 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: $\Rightarrow$ in $\lambda$ is not terminating, but confluent
$\Rightarrow$ in $\lambda$ is terminating and confluent, i.e. convergent

Problem: is a given set of reduction rules terminating?
- undecidable
When is \( \rightarrow \) Terminating?

**Basic Idea**: when the \( r_i \) are in some way simpler than the \( l_i \).

More formally: \( \rightarrow \) is terminating when there is a well founded order \( < \) in which \( r_i < l_i \) for all rules.

(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:

\[
s < t \iff \text{size}(s) < \text{size}(t) \quad \text{with}
\]
\[
\text{size}(s) = \text{number of function symbols in } s
\]

\[\exists\ g\ x < r\ f(\ g\ x) \text{ and } f\ x < r\ g(\ f\ x)\]

\[<\text{ is well founded, because }<\text{ is well founded on } \mathbb{N}\]

---

**Term Rewriting in Isabelle**

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

- \( \text{apply simp} \)
  - uses simplification rules
  - (almost) blindly from left to right
  - until no rule is applicable.

**Control**

- Equations turned into simplification rules with \([\text{simp}]\) attribute
- Adding/deleting equations locally:
  - \( \text{apply (simp add:<rules>) and apply (simp del:<rules>)} \)
- Using only the specified set of equations:
  - \( \text{apply (simp only:<rules>)} \)

---

**DEMO**