Slide 1

Theorem Proving
Principles, Techniques, Applications

Slide 2

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

LAST TIME ON HOL

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

Slide 3

LAST TIME ON HOL

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

Slide 4

THE THREE BASIC WAYS OF INTRODUCING THEOREMS

- Axioms:
  Example: axioms refl: "t = t"
  Do not use. Evil. Can make your logic inconsistent.

- Definitions:
  Example: defs inj_def: "inj f ≡ ∀x y. f x = f y → x = y"

- Proofs:
  Example: lemma "inj (λx. x + 1)"
  The harder, but safe choice.

THE THREE BASIC WAYS OF INTRODUCING TYPES

- Axioms:
  Example: axioms refl: "t = t"
  Do not use. Evil. Can make your logic inconsistent.

- Definitions:
  Example: defs inj_def: "inj f ≡ ∀x y. f x = f y → x = y"

- Proofs:
  Example: lemma "inj (λx. 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 is non-empty.
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:
- \( \mathbf{Mathematics} \) (algebra, group theory, etc)
- \( \mathbf{Functional~Programming} \) (model of execution)
- \( \mathbf{Theorem~Proving} \) (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 \( l \rightarrow r \)
Arrow Cheat Sheet

\[ \begin{align*}
0 &\rightarrow \{ (x,y) \mid x = y \} & \text{identity} \\
\overset{n+1}{\rightarrow} &\rightarrow \{ (x,y) \mid x = y \} & \text{n+1 fold composition} \\
\overset{i}{\rightarrow} &\rightarrow \cup_{i \geq 0} \overset{i}{\rightarrow} & \text{transitive closure} \\
\overset{+}{\rightarrow} &\rightarrow \cup_{i \geq 0} \overset{i}{\rightarrow} & \text{reflexive transitive closure} \\
\overset{n+1}{\rightarrow} &\rightarrow \cup_{i \geq 0} \overset{i}{\rightarrow} & \text{reflexive closure} \\
\overset{-1}{\rightarrow} &\rightarrow \{ (y,x) \mid x = y \} & \text{inverse} \\
\overset{\rightarrow}{\rightarrow} &\rightarrow \cup_{i \geq 0} \overset{i}{\rightarrow} & \text{symmetric closure} \\
\overset{+}{\rightarrow} &\rightarrow \cup_{i \geq 0} \overset{i}{\rightarrow} & \text{transitive symmetric closure} \\
\overset{+}{\rightarrow} &\rightarrow \cup_{i \geq 0} \overset{i}{\rightarrow} & \text{reflexive transitive symmetric closure}
\end{align*} \]

How to Decide \( l \overset{*}{\rightarrow} r \)

Same idea as for \( \beta \): look for \( n \) such that \( l \overset{n}{\rightarrow} n \) and \( r \overset{n}{\rightarrow} n \).

Does this always work?

- If \( l \overset{n}{\rightarrow} n \) and \( r \overset{n}{\rightarrow} n \) then \( l \overset{*}{\rightarrow} r \). Ok.
- If \( l \overset{*}{\rightarrow} r \), will there always be a suitable \( n \)? No!

Example:

Rules:
- \( f \ x \rightarrow a \), \( g \ x \rightarrow b \), \( f \ (g \ x) \rightarrow b \)
- \( f \ x \overset{\rightarrow}{\rightarrow} g \ x \) because \( f \ x \rightarrow a \overset{\rightarrow}{\rightarrow} f \ (g \ x) \rightarrow b \overset{\rightarrow}{\rightarrow} g \ x \)
- But: \( f \ x \rightarrow a \) and \( g \ x \rightarrow b \) and \( a, b \) in normal form

Works only for systems with Church-Rosser property:

\[ l \overset{*}{\rightarrow} r \Rightarrow \exists n, l \overset{n}{\rightarrow} n \land r \overset{n}{\rightarrow} n \]

Fact: \( \overset{*}{\rightarrow} \) is Church-Rosser iff it is confluent.

Confluence

Problem:

- is a given set of reduction rules confluent?

\[ \text{undecidable} \]

Local Confluence

Fact: local confluence and termination \( \Rightarrow \) confluence

Termination

\[ \overset{*}{\rightarrow} \]

- 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:

\[ \overset{\rightarrow}{\rightarrow} \] in \( \lambda \) is not terminating, but confluent
\[ \overset{\rightarrow}{\rightarrow} \] in \( \lambda \) is terminating and confluent, i.e. convergent

Problem: is a given set of reduction rules terminating?

\[ \text{undecidable} \]
**When is → Terminating?**

**Basic Idea:** when the $r_i$ are in some way simpler than the $l_i$

**More formally:** → is terminating when there is a well-founded order $<$ in which $r_i < l_i$ for all rules. (well-founded $\equiv$ no infinite decreasing chains $a_1 > a_2 > \ldots$)

**Example:** $f\ (g\ x) → g\ x,\ g\ (f\ x) → f\ x$

This system always terminates. Reduction order:

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

1. $g\ x <_r f\ (g\ x)$ and $f\ x <_r g\ (f\ x)$
2. $<_r$ is well-founded, because $<$ 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.

**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:
  ```text
  apply (simp add: <rules>) and apply (simp del: <rules>)
  ```

→ Using only the specified set of equations:
  ```text
  apply (simp only: <rules>)
  ```

**Demo**
ISAR

A LANGUAGE FOR STRUCTURED PROOFS

---

**ISAR**

- apply scripts
  - unreadable → Elegance?
  - hard to maintain → Explaining deeper insights?
  - do not scale → Large developments?

No structure. ISar!

---

A TYPICAL ISAR PROOF

```
proof
  assume formula₀
  have formula₁ by simp
  ...
  have formulaₙ by blast
  show formulaₙ₊₁ by ...
qed
```

proves \( \text{formula}_0 \implies \text{formula}_{n+1} \)

(analogous to assumes/shows in lemma statements)

---

ISAR CORE SYNTAX

```
proof = proof [method] statement^ qed
  \[ by \ method \]

method = (simp ...) | (blast ...) | (rule ...) | ...

statement = fix variables (\wedge)
  \[ assume \ proposition \ (\implies) \]
  \[ [from \ name^\*] \ (have | show) \ proposition \ proof \]
  \[ next \]

(proposition = [name:] formula (separates subgoals))
```
**PROOF AND QED**

proof [method] statement qed

lemma "[A; B] \implies A \land B"
proof (rule conjI)
  assume A: "A"
  from A show "A" by assumption
next
  assume B: "B"
  from B show "B" by assumption
qed

→ proof (<method>) applies method to the stated goal
→ proof applies a single rule that fits
→ proof - does nothing to the goal

**THE THREE MODES OF ISAR**

→ [prove]:
goal has been stated, proof needs to follow.

→ [state]:
proof block has openend or subgoal has been proved,
new from statement, goal statement or assumptions can follow.

→ [chain]:
from statement has been made, goal statement needs to follow.

**HOW DO I KNOW WHAT TO ASSUME AND SHOW?**

Look at the proof state!

lemma "[A; B] \implies A \land B"
proof (rule conjI)

→ proof (rule conjI) changes proof state to
  1. [A; B] \implies A
  2. [A; B] \implies B
→ so we need 2 shows: show "A" and show "B"
→ We are allowed to assume A,
  because A is in the assumptions of the proof state.

**THE THREE MODES OF ISAR**

→ [prove]:
goal has been stated, proof needs to follow.

→ [state]:
proof block has openend or subgoal has been proved,
new from statement, goal statement or assumptions can follow.

→ [chain]:
from statement has been made, goal statement needs to follow.

**HVE**

Can be used to make intermediate steps.

**Example:**

lemma "(x :: nat) + 1 = 1 + x"
proof -
  have A: "x + 1 = Suc x" by simp
  have B: "1 + x = Suc x" by simp
  show "x + 1 = 1 + x" by (simp only: A B)
qed
WE HAVE LEARNED TODAY …

- Introducing new Types
- Equations and Term Rewriting
- Confluence and Termination of reduction systems
- Term Rewriting in Isabelle
- First structured proofs (Isar)

EXERCISES

- use typedef to define a new type \( v \) with exactly one element.
- define a constant \( u \) of type \( v \)
- show that every element of \( v \) is equal to \( u \)
- design a set of rules that turns formulae with \( \land, \lor, \rightarrow, \neg \)
  into disjunctive normal form
  (= disjunction of conjunctions with negation only directly on variables)
- prove those rules in Isabelle
- use simp only with these rules on \( \neg B \rightarrow C \rightarrow A \rightarrow B \)