Theorem Proving
Principles, Techniques, Applications

Slide 1

Slide 2

Content

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

Slide 3

λ calculus is inconsistent

From last lecture:
Can find term $R$ such that $R \Rightarrow \neg(R R)$

There are more terms that do not make sense:
$12$, $true false$, etc.

Solution: rule out ill-formed terms by using types.
(Church 1940)

Slide 4

Introducing types

Idea: assign a type to each “sensible” λ term.

Examples:

→ for term $t$ has type $\alpha$ write $t :: \alpha$
→ if $x$ has type $\alpha$ then $\lambda x. x$ is a function from $\alpha$ to $\alpha$
   Write: $(\lambda x. x) :: \alpha \Rightarrow \alpha$
→ for $s \tau$ to be sensible:
   $s$ must be function
   $\tau$ must be right type for parameter
   If $s :: \alpha \Rightarrow \beta$ and $t :: \alpha$ then $(s \ t) :: \beta$
That’s about it

Now formally, again

Syntax for $\lambda^-$

Terms: $t ::= v | c | (t \; t) | (\lambda x. \; t)$

$v, x \in \mathbf{V}, \; c \in \mathbf{C}, \; \mathbf{V, C}$ sets of names

Types: $\tau ::= \mathbf{b} | \nu \mid \tau \Rightarrow \tau$

$\mathbf{b} \in \{\text{bool}, \text{int}, \ldots\}$ base types

$\nu \in \{\alpha, \beta, \ldots\}$ type variables

$\alpha \Rightarrow \beta \Rightarrow \gamma = \alpha \Rightarrow (\beta \Rightarrow \gamma)$

Contexts $\Gamma$:

$\Gamma$: function from variable and constant names to types.

Term $t$ has type $\tau$ in context $\Gamma$:

$\Gamma \vdash t :: \tau$

Examples

$\Gamma \vdash (\lambda x. \; x) :: \alpha \Rightarrow \alpha$

$[y \leftarrow \text{int}] \vdash y :: \text{int}$

$[z \leftarrow \text{bool}] \vdash (\lambda y. \; y) \; z :: \text{bool}$

$[] \vdash \lambda f. \; f \; x :: (\alpha \Rightarrow \beta) \Rightarrow \alpha \Rightarrow \beta$

A term $t$ is well typed or type correct

if there are $\Gamma$ and $\tau$ such that $\Gamma \vdash t :: \tau$
**Type Checking Rules**

Variables: \[ \Gamma \vdash x :: \Gamma(x) \]

Application: \[ \begin{array}{l} \Gamma \vdash t_1 :: \tau_1, \Gamma \vdash t_2 :: \tau_2 \\ \Gamma \vdash (t_1 \, t_2) :: \tau_1 \end{array} \]

Abstraction: \[ \Gamma[x \leftarrow \tau_1] \vdash t :: \tau_2 \]
\[ \Gamma \vdash (\lambda x. \, t) :: \tau_1 \Rightarrow \tau_2 \]

**Example Type Derivation:**

\[ \begin{array}{l} [x \leftarrow \alpha, y \leftarrow \beta] \vdash x :: \alpha \\ [x \leftarrow \alpha] \vdash \lambda y. \, x :: \beta \Rightarrow \alpha \\ [\lambda \vdash \lambda x \, y. \, x :: \alpha \Rightarrow \beta \Rightarrow \alpha] \end{array} \]

**More Complex Example**

\[ \begin{array}{l} \Gamma \vdash f :: \alpha \Rightarrow (\alpha \Rightarrow \beta) \\ \Gamma \vdash x :: \alpha \\ \Gamma \vdash \lambda f. \, f \, x :: \beta \\ \Gamma \vdash \lambda x. \, f \, x \, x :: (\alpha \Rightarrow \beta) \Rightarrow \alpha \Rightarrow \beta \\ \Gamma \vdash [f \leftarrow \alpha \Rightarrow \beta, x \leftarrow \alpha] \end{array} \]

**More General Types**

A term can have more than one type.

Example: \[ [] \vdash \lambda x. \, x :: \text{bool} \Rightarrow \text{bool} \]
\[ [] \vdash \lambda x. \, x :: \alpha \Rightarrow \alpha \]

Some types are more general than others:

\[ \tau \leq \sigma \quad \text{if there is a substitution } S \text{ such that } \tau = S(\sigma) \]

Examples:

\[ \text{int} \Rightarrow \text{bool} \leq \alpha \Rightarrow \beta \leq \beta \Rightarrow \alpha \leq \alpha \Rightarrow \alpha \]

**More Complex Example**

5

**Most General Types**

6


**MOST GENERAL TYPES**

**Fact:** each type correct term has a most general type

**Formally:**
\[
\Gamma \vdash t : \sigma \implies \exists \sigma'. \Gamma \vdash t : \sigma \land (\forall \sigma'. \Gamma \vdash t : \sigma' \implies \sigma' \subseteq \sigma)
\]

It can be found by executing the typing rules backwards.

- **type checking:** checking if \( \Gamma \vdash t : \sigma \) for given \( \Gamma \) and \( \sigma \)
- **type inference:** computing \( \Gamma \) and \( \sigma \) such that \( \Gamma \vdash t : \sigma \)

Type checking and type inference on \( \lambda^- \) are decidable.

**WHAT ABOUT \( \beta \) REDUCTION?**

Definition of \( \beta \) reduction stays the same.

**Fact:** Well typed terms stay well typed during \( \beta \) reduction

**Formally:**
\[
\Gamma \vdash s : \tau \land s \rightarrow_{\beta} t \implies \Gamma \vdash t : \tau
\]

This property is called **subject reduction**

**WHAT ABOUT TERMINATION?**

\( \beta \) reduction in \( \lambda^- \) always terminates.

(Alan Turing, 1942)

- **\( \Rightarrow_{\beta} \)** **is decidable**
  
  To decide if \( s \Rightarrow_{\beta} t \), reduce \( s \) and \( t \) to normal form (always exists, because \( \rightarrow_{\beta} \) terminates), and compare result.

- **\( \Rightarrow_{\eta} \)** **is decidable**
  
  This is why Isabelle can automatically reduce each term to \( \beta \eta \) normal form.

**WHAT DOES THIS MEAN FOR EXPRESSIVENESS?**

Not all computable functions can be expressed in \( \lambda^- \)!

How can typed functional languages then be turing complete?

**Fact:**

Each computable function can be encoded as closed, type correct \( \lambda^- \) term using \( Y : (\tau \Rightarrow \tau) \Rightarrow \tau \) with \( Y t \rightarrow_{\beta} t (Y t) \) as only constant.

- \( Y \) is called fix point operator
- used for recursion
Types and Terms in Isabelle

Types: \[ \tau ::= \mathbf{b} \mid 'v \mid 'v : C \mid \tau \Rightarrow \tau \mid (\tau, \ldots, \tau) K \]
\( \mathbf{b} \in \{\text{bool, int, ...}\} \) base types
\( \nu \in \{\alpha, \beta, \ldots\} \) type variables
\( K \in \{\text{set, list, ...}\} \) type constructors
\( C \in \{\text{order, linord, ...}\} \) type classes

Terms: \[ t ::= v \mid c \mid ?v \mid (tt) \mid (\lambda x. t) \]
\( v, x \in V, \ c \in C, \ V, C \) sets of names

- **type constructors**: construct a new type out of a parameter type.
  Example: \( \text{int list} \)
- **type classes**: restrict type variables to a class defined by axioms.
  Example: \( \alpha : \text{order} \)
- **schematic variables**: variables that can be instantiated.

Type Classes

- Similar to Haskell’s type classes, but with semantic properties
  - axclass order < ord
  - order_refl: "\( x \leq x " \)
  - order_trans: "\( [x \leq y; y \leq z] \Rightarrow x \leq z " \)
  ...

  - theorems can be proved in the abstract
    - lemma order_lemma: "\( \exists x :: \alpha : \text{order}. \ [x < y; y < z] \Rightarrow x < z " \)
  - can be used for subtyping
    - axclass linorder < order
    - linorder_linear: "\( x \leq y \lor y \leq x " \)
  - can be instantiated
    - instance nat :: "\( \{\text{order, linorder}\} \) " by ...

Schematic Variables

- \( X \land Y \)
- \( X \) and \( Y \) must be instantiated to apply the rule

  But: lemma "\( x + 0 = 0 + x " \)

- \( x \) is free
- convention: lemma must be true for all \( x \)
- during the proof, \( x \) must not be instantiated

Solution:
Isabelle has free (\( x \)), bound (\( x \)), and schematic (?X) variables.

Only schematic variables can be instantiated.
Free converted into schematic after proof is finished.

Higher Order Unification

Unification:
Find substitution \( \sigma \) on variables for terms \( s, t \) such that \( \sigma(s) = \sigma(t) \)

In Isabelle:
Find substitution \( \sigma \) on schematic variables such that \( \sigma(s) =_{\alpha, \beta, \gamma} \sigma(t) \)

Examples:
- \( ?X \land ?Y \quad \alpha \beta \gamma \quad x \land x \quad [?X \leftarrow x, ?Y \leftarrow x] \)
- \( ?P \ x \quad \alpha \beta \gamma \quad x \land x \quad [?P \leftarrow \lambda x. x \land x] \)
- \( P \ (?f \ x) \quad \alpha \beta \gamma \quad ?Y \ x \quad [?f \leftarrow \lambda x. x, ?Y \leftarrow P] \)

Higher Order: schematic variables can be functions.
HIGHER ORDER UNIFICATION

- Unification modulo $\alpha\beta$ (Higher Order Unification) is semi-decidable
- Unification modulo $\alpha\beta\eta$ is undecidable
- Higher Order Unification has possibly infinitely many solutions

**But:**
- Most cases are well-behaved
- Important fragments (like Higher Order Patterns) are decidable

**Higher Order Pattern:**
- is a term in $\beta$ normal form where
- each occurrence of a schematic variable is of the form $?f \, t_1 \ldots t_n$
- and the $t_1 \ldots t_n$ are $\eta$-convertible into $n$ distinct bound variables

---

WE HAVE LEARNED SO FAR...

- Simply typed lambda calculus: $\lambda^-$
- Typing rules for $\lambda^-$, type variables, type contexts
- $\beta$-reduction in $\lambda^-$ satisfies subject reduction
- $\beta$-reduction in $\lambda^-$ always terminates
- Types and terms in Isabelle

---

PREVIEW: PROOFS IN ISABELLE

**General schema:**

```
lemma name: "<goal>"
apply <method>
apply <method>
done
```

- Sequential application of methods until all subgoals are solved.
**THE PROOF STATE**

1. $\bigwedge x_1 \ldots x_p \cdot [A_1; \ldots; A_n] \Rightarrow B$
2. $\bigwedge y_1 \ldots y_q \cdot [C_1; \ldots; C_m] \Rightarrow D$

**ISABELLE THEORIES**

**Syntax:**

```plaintext
theory MyTh = ImpTh_1 + \ldots + ImpTh_n:
  (declarations, definitions, theorems, proofs, \ldots)
end
```

- $MyTh$: name of theory. Must live in file $MyTh.thy$
- $ImpTh_i$: name of imported theories. Import transitive.

Unless you need something special:

```plaintext
theory MyTh = Main:
```

**NATURAL DEDUCTION RULES**

**Slide 27**

- **Introduction** and **elimination** rules

**Slide 25**

- Parameters
- Local assumptions
- Actual (sub)goal

**NATURAL DEDUCTION RULES**

- **Conjunction** ($\wedge$):
  - $A \wedge B \Rightarrow \vdash C$
  - $A \wedge B \Rightarrow \vdash C$

- **Disjunction** ($\vee$):
  - $A \vee B \Rightarrow \vdash C$
  - $A \vee B \Rightarrow \vdash C$

- **Implication** ($\Rightarrow$):
  - $A \Rightarrow B \Rightarrow \vdash C$
  - $A \Rightarrow B \Rightarrow \vdash C$

**Slide 28**

- **Proof by assumption**
  - **apply** assumption

- **Backtracking!**
  - Backtracking command: **back**
**Intro Rules**

Intro rules decompose formulae to the right of $\implies$.

**apply** (rule <intro-rule>)

Intro rule \( [A_1; \ldots; A_n] \implies A \) means

- To prove \( A \) it suffices to show \( A_1 \ldots A_n \)

Applying rule \( [A_1; \ldots; A_n] \implies A \) to subgoal \( C \):

- unify \( A \) and \( C \)
- replace \( C \) with \( n \) new subgoals \( A_1 \ldots A_n \)

**Elim Rules**

Elim rules decompose formulae on the left of $\implies$.

**apply** (erule <elim-rule>)

Elim rule \( [A_1; \ldots; A_n] \implies A \) means

- If I know \( A_1 \) and want to prove \( A \) it suffices to show \( A_2 \ldots A_n \)

Applying rule \( [A_1; \ldots; A_n] \implies A \) to subgoal \( C \):

Like rule but also

- unifies first premise of rule with an assumption
- eliminates that assumption

**Exercises**

- what are the types of \( \lambda x \ y \ x \) and \( \lambda x \ y \ z \ x \ y \ (y\ z) \)
- construct a type derivation tree on paper for \( \lambda x \ y \ z \ x \ y \ (y\ z) \)
- find a unifier (substitution) such that \( \lambda x \ y \ ?F \ x = \lambda x \ y \ c \ (y\ y) \)
- prove \( (A \rightarrow B \rightarrow C) = (A \wedge B \rightarrow C) \) in Isabelle
- prove \( \neg(A \wedge B) \implies \neg A \vee \neg B \) in Isabelle (tricky!)