ENOUGH THEORY!
GETTING STARTED WITH ISABELLE

E N O U G H  T H E O R Y !  
G E T T I N G  S T A R T E D  W I T H  I S A B E L L E

Slide 1

System Architecture

Proof General – user interface

HOL, ZF – object-logics

Isabelle – generic, interactive theorem prover

Standard ML – logic implemented as ADT

User can access all layers!

Slide 3

System Requirements

→ Linux, FreeBSD, MacOS X or Solaris
→ Standard ML
  (PolyML fastest, SML/NJ supports more platforms)
→ XEmacs or Emacs
  (for ProofGeneral)

If you have only Windows, try IsaMorph
http://www.brucker.ch/projects/isamorph/ or
install Cygwin.

Slide 4
Documentation

Available from http://isabelle.in.tum.de

• Learning Isabelle
  • Tutorial on Isabelle/HOL (LNCS 2283)
  • Tutorial on Isar
  • Tutorial on Locales

• Reference Manuals
  • Isabelle/Isar Reference Manual
  • Isabelle Reference Manual
  • Isabelle System Manual

• Reference Manuals for Object-Logics

ProofGeneral

• User interface for Isabelle
• Runs under XEmacs or Emacs
• Isabelle process in background

Interaction via
• Basic editing in XEmacs (with highlighting etc)
• Buttons (tool bar)
• Key bindings
• ProofGeneral Menu (lots of options, try them)

X-Symbol Cheat Sheet

Input of funny symbols in ProofGeneral

• via menu ("X-Symbol")
• via ASCII encoding (similar to \LaTeX\): \&\&, |\|, . . .
• via abbreviation: \&\& , |\| , . . .
• via rotate: \L , \rightarrow , \leftarrow , . . . (cycles through variations of letter)

<table>
<thead>
<tr>
<th>V</th>
<th>A</th>
<th>&amp;&amp;</th>
<th>|</th>
</tr>
</thead>
<tbody>
<tr>
<td>\forall</td>
<td>\exists</td>
<td>\lambda</td>
<td>\rightarrow</td>
</tr>
</tbody>
</table>

1 converted to X-Symbol
2 stays ASCII
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 9

\[ \lambda \text{-calculus} \]

Alonzo Church

- lived 1903–1995
- supervised people like Alan Turing, Stephen Kleene
- famous for Church-Turing thesis, lambda calculus,
  first undecidability results
- invented \( \lambda \) calculus in 1930’s

\[ \lambda \text{-calculus} \]

- originally meant as foundation of mathematics
- important applications in theoretical computer science
- foundation of computability and functional programming

Slide 10

untyped \( \lambda \)-calculus

- turing complete model of computation
- a simple way of writing down functions

Basic intuition:

Instead of \( f(x) = x + 5 \)

Write \( f = \lambda x. x + 5 \)

\( \lambda x. x + 5 \)

- a term
- a nameless function
- that adds 5 to its parameter

Slide 11

Function Application

For applying arguments to functions

Instead of \( f(x) \)

Write \( f \ x \)

Example: \( (\lambda x. x + 5) \ a \)

Evaluating: \( \text{in } (\lambda x. t) \ a \text{ replace } x \text{ by } a \text{ in } t \)

(computation)

Example: \( (\lambda x. x + 5) \ (a + b) \text{ evaluates to } (a + b) + 5 \)

Slide 12
Syntax

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

- \( v, x \) variables
- \( c \) constants
- \( (t t) \) application
- \( (\lambda x. t) \) abstraction

Conventions

- leave out parentheses where possible
- list variables instead of multiple \( \lambda \)

Example: instead of \( (\lambda y. (\lambda x. (x y))) \) write \( \lambda y. x \ y \)

Rules:
- list variables: \( \lambda x. (\lambda y. t) = \lambda x. y \ t \)
- application binds to the left: \( x \ y \ z = (x \ y) \ z \neq x \ (y \ z) \)
- abstraction binds to the right: \( \lambda x. x \ y = \lambda x. (x \ y) \neq (\lambda x. x) \ y \)
- leave out outermost parentheses
Getting used to the Syntax

Example:

\[ \lambda x \, y \, z. \, x \, z \]
\[ (y \, z) = \]
\[ \lambda x \, y \, z. \, (x \, z) \, (y \, z) = \]
\[ \lambda x \, y \, z. \, ((x \, z) \, (y \, z)) = \]
\[ \lambda x. \, \lambda y. \, \lambda z. \, ((x \, z) \, (y \, z)) = \]
\[ (\lambda x. \, (\lambda y. \, \lambda z. \, ((x \, z) \, (y \, z)))) \]

Computation

\[ (\lambda x. \, y. \, x. \, f \, (y \, x) \, 5) \] \[ \rightarrow \beta \]
\[ (\lambda y. \, f \, (y \, 5)) \] \[ \rightarrow \beta \]
\[ f \, (\lambda x. \, x) \, 5 \] \[ \rightarrow \beta \]
\[ f \, 5 \]

Defining Computation

\[ \beta \text{ reduction:} \]
\[ (\lambda x. \, s \, t) \rightarrow s [x \leftarrow t] \]
\[ (\lambda x. \, s) \, t \rightarrow s [t] \]
\[ (\lambda x. \, s) \, t \rightarrow s [t] \]
\[ (\lambda x. \, s) \, (\lambda y. \, s') \rightarrow (\lambda x. \, s) \, (\lambda y. \, s') \]

Still to do: define \( s [x \leftarrow t] \)

Defining Substitution

Easy concept. Small problem: variable capture.

Example: \( (\lambda x. \, x \, z) \] \[ \rightarrow \beta \]

We do not want: \( (\lambda x. \, x \, z) \] as result.

What do we want?

In \( (\lambda y. \, z) \, z \] \[ \rightarrow \beta \], there would be no problem.

So, solution is: rename bound variables.

Still to do: define \( s [x \leftarrow t] \)
Free Variables

Bound variables: in \( (\lambda x. t) \), \( x \) is a bound variable.

Free variables \( FV \) of a term:

\[
FV(x) = \{x\} \\
FV(c) = \{\} \\
FV(st) = FV(s) \cup FV(t) \\
FV(\lambda x. t) = FV(t) \setminus \{x\}
\]

Example: \( FV(\lambda x. (\lambda y. x) y x) = \{y\} \)

Term \( t \) is called closed if \( FV(t) = \{\} \)

Our problematic substitution example, \( (\lambda x. z)(x ← z) \), is problematic because the bound variable \( x \) is a free variable of the replacement term "z".

Substitution

\[
x[x ← t] = t \\
y[x ← t] = y \quad \text{if} \ x \neq y \\
e[x ← t] = e
\]

\[
(s_1.s_2)[x ← t] = (s_1[x ← t].s_2[x ← t]) \\
(\lambda x. s)[x ← t] = (\lambda x. s) \\
(\lambda y. s)[x ← t] = (\lambda y. s[x ← t]) \quad \text{if} \ x \neq y \text{ and } y \notin FV(t) \\
(\lambda y. s)[x ← t] = (\lambda z. s[y ← z][x ← t]) \quad \text{if} \ x \neq y \quad \text{and} \ z \notin FV(t) \cup FV(s)
\]

Substitution Example

\[
(x(\lambda x. (\lambda y. z x))(x ← y)) \\
= (x[x ← y])((\lambda x. x)(x ← y))((\lambda y. z x)(x ← y)) \\
= y(\lambda x. (\lambda y. z y))
\]

\( α \) Conversion

Bound names are irrelevant:
\( \lambda x. x \) and \( \lambda y. y \) denote the same function.

\( α \) conversion:
\( s =_α t \) means \( s = t \) up to renaming of bound variables.

Formally:

\[
\begin{align*}
(\lambda x. t) & \rightarrow_α (\lambda y. t[x ← y]) \text{ if } y \notin FV(t) \\
(\lambda x. s) & \rightarrow_α (\lambda y. s) \\
(s_1) & \rightarrow_α (s_2) \rightarrow_α (s_1) \\
(s_1.t) & \rightarrow_α (s_2.t) \rightarrow_α (s_1.t) \\
(s_1.s_2) & \rightarrow_α (s_1.s_2) \rightarrow_α (s_1.s_2) \\
\end{align*}
\]

\( s =_α t \) iff \( s \rightarrow_α^* t \)

\( (→_α^* = \text{transitive, reflexive closure of } →_α = \text{multiple steps}) \)
Equality in Isabelle is equality modulo $\alpha$ conversion:

if $s \equiv_{\alpha} t$ then $s$ and $t$ are syntactically equal.

Examples:

- $x \ (\lambda x \ y. x y) \equiv_{\alpha} x \ (\lambda y \ x. y x)$
- $x \ (\lambda z \ y. z y) \not\equiv_{\alpha} x \ (\lambda x \ x. x x)$

Does every $\lambda$ term have a normal form?

No!

Example:

$(\lambda x \ y. y) \ ((\lambda x \ x) \ (\lambda x \ x)) \not\rightarrow_{\beta}$

$\lambda$ calculus is not terminating

$\beta$ reduction is confluent

Confluence: $s \rightarrow_{\beta} t \wedge s \rightarrow_{\beta} u \Rightarrow \exists t. x \rightarrow_{\beta} t \wedge y \rightarrow_{\beta} t$

Order of reduction does not matter for result

Normal forms in $\lambda$ calculus are unique
\( \beta \) reduction is confluent

Example:

\((\lambda x \ y. \ y) \ ((\lambda x. \ x \ x) \ a) \rightarrow_\beta (\lambda x \ y. \ y) \ (a \ a) \rightarrow_\beta \lambda y. \ y\)

\((\lambda x \ y. \ y) \ ((\lambda x. \ x \ x) \ a) \rightarrow_\beta \lambda y. \ y\)

\( \eta \) Conversion

Another case of trivially equal functions: \( t = (\lambda x. \ t \ x) \)

Definition:

\[
\begin{align*}
\lambda x. t & \rightarrow_\eta t & \text{if } x \notin \text{FV}(t) \\
(s \ t) & \rightarrow_\eta (s' \ t) \\
(\lambda x. \ s) & \rightarrow_\eta (\lambda x. \ s') \\
s & \rightarrow_\eta t & \text{iff } \exists n. s \rightarrow^*_\eta n \land t \rightarrow^*_\eta n
\end{align*}
\]

Example: \((\lambda x. \ f \ x) \ (\lambda y. \ g \ y) \rightarrow_\eta (\lambda x. \ f \ x) \ g \rightarrow_\eta f \ g\)

\( \eta \) reduction is confluent and terminating.

\( \rightarrow_\eta \) is confluent.

\( \rightarrow_\eta \) means \( \rightarrow_\beta \) and \( \rightarrow_\eta \) steps are both allowed.

Equality in Isabelle is also modulo \( \eta \) conversion.

In fact ...

Equality in Isabelle is modulo \( \alpha, \beta, \) and \( \eta \) conversion.

We will see later why that is possible.

Exercises

\( \rightarrow_\eta \) reduction is confluent and terminating.

\( \rightarrow_\eta \) means \( \rightarrow_\beta \) and \( \rightarrow_\eta \) steps are both allowed.

Equality in Isabelle is also modulo \( \eta \) conversion.