ENOUGH THEORY!
GETTING STARTED WITH ISABELLE
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{}): \&and, \lor, ...
- via abbreviation: \&\&, \lor, ...
- via rotate: \<-, \rightarrow (cycles through variations of letter)

<table>
<thead>
<tr>
<th>Symbol</th>
<th>ASCII</th>
<th>X-Symbol</th>
</tr>
</thead>
<tbody>
<tr>
<td>\forall</td>
<td>\forall</td>
<td>\forall</td>
</tr>
<tr>
<td>\exists</td>
<td>\exists</td>
<td>\exists</td>
</tr>
<tr>
<td>\lambda</td>
<td>\lambda</td>
<td>\lambda</td>
</tr>
<tr>
<td>\not</td>
<td>\not</td>
<td>\not</td>
</tr>
<tr>
<td>\land</td>
<td>&amp;</td>
<td>&amp;</td>
</tr>
<tr>
<td>\lor</td>
<td>|</td>
<td>|</td>
</tr>
<tr>
<td>\rightarrow</td>
<td>\rightarrow</td>
<td>\rightarrow</td>
</tr>
</tbody>
</table>

1. converted to X-Symbol
2. stays ASCII

DEMO
**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

---

**Untyped \( \lambda \)-Calcululus**

- 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

---

**Function Application**

For applying arguments to functions

Instead of \( f(x) \)

Write \( f x \)

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

**Evaluating:**

In \( (\lambda x. t) a \) replace \( x \) by \( a \) in \( t \) (computation!)

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

---

**\( \lambda \)-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

- Originally meant as foundation of mathematics
- Important applications in theoretical computer science
- Foundation of computability and functional programming
THAT’S IT!

NOW FORMAL

SYNTAX

Terms: \[ t ::= v | c | (t t) | (\lambda x. t) \]

\[ v, x \in V, \; c \in C, \; V, C \text{ sets of names} \]

- \( v, x \) 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:

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

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

---

**Computation**

Intuition: replace parameter by argument
this is called \(\beta\)-reduction

Example

\[ (\lambda x\ y\ .\ f\ (y\ x))\ 5 \rightarrow_{\beta} (\lambda x\ .\ x) \]

\[ (\lambda y\ .\ f\ (y\ 5))\ (\lambda x\ .\ x) \rightarrow_{\beta} f\ ((\lambda x\ .\ x)\ 5) \rightarrow_{\beta} f\ 5 \]

---

**Defining Computation**

\(\beta\) reduction:

\[ (\lambda x\ .\ s)\ t \rightarrow_{\beta} s[x \leftarrow t] \]

\[ s \rightarrow_{\beta} s' \quad \Rightarrow \quad (s\ t) \rightarrow_{\beta} (s'\ t) \]

\[ t \rightarrow_{\beta} t' \quad \Rightarrow \quad (s\ t) \rightarrow_{\beta} (s\ t') \]

\[ s \rightarrow_{\beta} s' \quad \Rightarrow \quad (\lambda x\ .\ s) \rightarrow_{\beta} (\lambda x\ .\ s') \]

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

---

**Defining Substitution**

Easy concept. Small problem: variable capture.

Example: \((\lambda x\ .\ x)\ [z \leftarrow z]\)

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

What do we want?

In \((\lambda y\ .\ y\ z)\ [z \leftarrow x] = (\lambda y\ .\ y\ z)\) there would be no problem.

So, solution is: rename bound variables.
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: \(FV(\lambda x.(\lambda y.(\lambda x. y) y) x) = \{y\}\)

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

SUBSTITUTION

\[
\begin{align*}
  x \left[ x \leftarrow t \right] &= t \\
  y \left[ x \leftarrow t \right] &= y & \text{if } x \neq y \\
  c \left[ x \leftarrow t \right] &= c \\
  (s_1 s_2) \left[ x \leftarrow t \right] &= (s_1[x \leftarrow t] s_2[x \leftarrow t]) \\
  (\lambda x. s) \left[ x \leftarrow t \right] &= (\lambda x. s) \\
  (\lambda y. s) \left[ x \leftarrow t \right] &= (\lambda y. s[x \leftarrow t]) & \text{if } x \neq y \text{ and } y \notin FV(t) \\
  (\lambda y. s) \left[ x \leftarrow t \right] &= (\lambda z. s[y \leftarrow z][x \leftarrow t]) & \text{if } x \neq y \text{ and } z \notin FV(t) \cup FV(s)
\end{align*}
\]

\(\alpha\) CONVERSION

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

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

Formally:

\[
\begin{align*}
  s \rightarrow_\alpha t' & \quad \text{iff } s \rightarrow^\ast_\alpha t' \\
  (\lambda x. s) \rightarrow_\alpha (\lambda y. t) & \quad \text{iff } y \notin FV(t) \\
  (s t) \rightarrow_\alpha (s' t') & \quad \text{iff } s \rightarrow_\alpha s' \\
  (s t) \rightarrow_\alpha (s' t) & \quad \text{iff } t \rightarrow_\alpha t' \\
  s \rightarrow_\alpha s' & \quad \text{iff } (s \rightarrow^\ast_\alpha s') \\
  s \rightarrow^\ast_\alpha t & \quad \text{iff } (\cdots \rightarrow^\ast_\alpha t)
\end{align*}
\]

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

if $s \alpha = t$ then $s$ and $t$ are syntactically equal.

Examples:

- $x (\lambda x y. x y) \alpha = x (\lambda y x. y x)$
- $x (\lambda y z. y y) \alpha \neq x (\lambda x x. x x)$

We have defined $\beta$ reduction: $\rightarrow_{\beta}$

Some notation and concepts:

- $\alpha$ conversion: $s \alpha \rightarrow t \iff \exists n. s \rightarrow^\ast \beta n \land t \rightarrow^\ast \beta n$
- $t$ is reducible if there is an $s$ such that $t \rightarrow_{\beta} s$
- $(\lambda x. s)$ is called a redex (reducible expression)
- $t$ is reducible iff it contains a redex
- if it is not reducible, $t$ is in normal form

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

No!

Example:

$$(\lambda x. x x) (\lambda x. x x) \rightarrow_{\beta} (\lambda x. x x) (\lambda x. x x) \rightarrow_{\beta} (\lambda x. x x) (\lambda x. x x) \rightarrow_{\beta} \ldots$$

(but: $(\lambda x y. y) ((\lambda x. x x) (\lambda x. x x)) \rightarrow_{\beta} \lambda y. y$)

$\lambda$ calculus is not terminating

$\beta$ reduction is confluent

Confluence: $s \rightarrow^\ast \beta x \land s \rightarrow^\ast \beta y \implies \exists t. x \rightarrow^\ast \beta t \land y \rightarrow^\ast \beta t$

Order of reduction does not matter for result

Normal forms in $\lambda$ calculus are unique
**β Reduction is Confluent**

Example:

\[(\lambda x\ y\ y)\ ((\lambda x\ x)\ a) \rightarrow^\beta (\lambda x\ y\ y)\ (a\ a) \rightarrow^\beta \lambda y\ y\]

**η Conversion**

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

Definition:

- \(s \rightarrow^\eta t\) if \(x \notin \text{FV}(t)\)
- \(s \rightarrow^\eta t\) if \(\exists n. s \rightarrow^* n \land t \rightarrow^* n\)

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.
- \(\eta\) is confluent.
- \(\eta\) 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 next lecture why that is possible.

**Exercises**

- Download and install Isabelle from https://isabelle.in.tum.de or http://mirror.cse.unsw.edu.au/pub/isabelle/
- Switch on X-Symbol in ProofGeneral
- Step through the demo files from the lecture web page
- Write an own theory file, look at some theorems in the library, try 'find theorem'