COMP4161
Advanced Topics in Software Verification

Thomas Sewell, Miki Tanaka, Rob Sison

T3/2024

Binary Search (`java.util.Arrays`)

```
1:      public static int binarySearch(int[] a, int key) {
2:          int low = 0;
3:          int high = a.length - 1;
4:
5:          while (low <= high) {
6:              int mid = (low + high) / 2;
7:              int midVal = a[mid];
8:
9:              if (midVal < key)
10:                  low = mid + 1
11:              else if (midVal > key)
12:                  high = mid - 1;
13:              else
14:                  return mid; // key found
15:          }
16:          return -(low + 1);  // key not found.
17:     }
```

```
6:              int mid = (low + high) / 2;
```

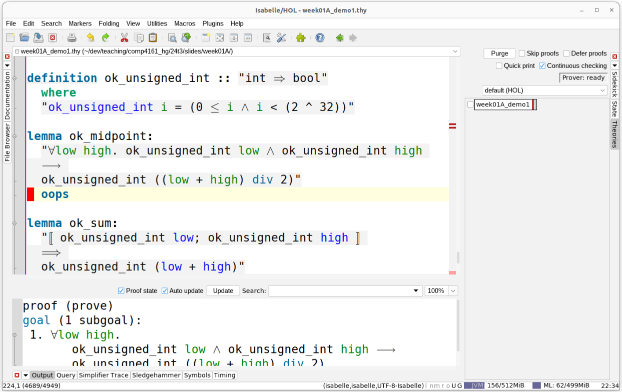
How can we fix tricky bugs like this?

```
6:            int mid = (low + high) / 2;
```

One approach is to *prove* our program *implementation* correct.

We can do this proof using a *theorem prover*.

- ➜ a system for checking proofs
- ➜ implemented in software



We’ll see this interactively soon.

What you will learn in COMP4161

- ➜ how to use a theorem prover
- ➜ how a theorem prover is built
- ➜ how to prove and specify
- ➜ how to reason about programs

This is what we (Rob, Miki & myself) do in our research work.

Health Warning

Theorem Proving is addictive

Organisation & Tutorials

When	Where	
Mon	12:00h - 14:00h	Science & Engineering G07 (K-E8-G07)
Wed	12:00h - 14:00h	Rupert Myers Theatre (K-M15-1001)

There are no separate tutorials. There will (obviously) be a break in the 12-2 lectures.

`http://www.cse.unsw.edu.au/~cs4161/`

Prerequisites

This is an advanced course. It assumes knowledge in

- ➜ Functional programming
- ➜ First-order formal logic

The following program should make sense to you:

```
map f []         =  []
map f (x : xs)   =  f x : map f xs
```

You should be able to read and understand this formula:

$$\exists x.\ (P(x) \longrightarrow \forall x.\ P(x))$$

Content — Using Theorem Provers

Rough timeline

➜ Theorem Proving: Foundations & Principles

- Intro, Lambda calculus, natural deduction [1,2]
- Higher Order Logic, Isar (part 1) [2,3[a]]
- Term rewriting [3,4]

➜ Proof & Specification Techniques

- Inductively defined sets, rule induction [4,5]
- Datatype induction, primitive recursion [5,7]
- General recursive functions, termination proofs [7[b]]
- Proof automation, Isar (part 2) [8]
- Hoare logic, proofs about programs, invariants [8,9]
- C verification [9,10]
- Practice, questions, exam prep [10[c]]

[a] a1 due; [b] a2 due; [c] a3 due

Miki: 1.2 $\to$ 3, Rob: 4 $\to$ 7.1, Thomas: 7.2 $\to$ 10

Interactive Proving

Isabelle is an *interactive* theorem prover.

➜ The user guides the tool, step by step if necessary.

This allows us to approach theory experimentally.

- Is it even theory any more?
- It feels different, and can be addictive.

Interacting with Isabelle is essential to this course.

- Large parts of the lectures will be interactive demos.
- We will train you to experiment and learn from the prover.
- You will get much more feedback on your proofs than in other theory assignments.

Things to do & not do to succeed in COMP4161

you should:

- ➜ attend lectures as much as you can
  - ➜ and be interactive!
- ➜ try Isabelle early
- ➜ redo the demos *yourself*
- ➜ try the exercises/homework we give

you should not:

- ➜ just read the slides
- ➜ commit **PLAGIARISM**
  - Assignments and exams are take-home. This does NOT mean you can work in groups. Each submission is personal.
  - For more info, see Plagiarism Policy[a]

[a] `https://student.unsw.edu.au/plagiarism`

Credits

on the topic of plagiarism, some material shamelessly stolen from



Tobias Nipkow, Larry Paulson, Markus Wenzel


David Basin, Burkhardt Wolff

These slides largely the work of past lecturers Gerwin Klein, June Andronick, Ramana Kumar, Toby Murray, Christine Rizkallah, Johannes Åman Pohjola.

What is a formal proof?

A derivation in a formal calculus

Example: $A \wedge B \longrightarrow B \wedge A$ is derivable in the following system

Rules: $\dfrac{X \in S}{S \vdash X}$ (assumption) $\quad \dfrac{S \cup \{X\} \vdash Y}{S \vdash X \longrightarrow Y}$ (impI)

$\dfrac{S \vdash X \quad S \vdash Y}{S \vdash X \wedge Y}$ (conjI) $\quad \dfrac{S \cup \{X, Y\} \vdash Z}{S \cup \{X \wedge Y\} \vdash Z}$ (conjE)

Proof:

1.	$\{A, B\} \vdash B$	(by assumption)
2.	$\{A, B\} \vdash A$	(by assumption)
3.	$\{A, B\} \vdash B \wedge A$	(by conjI with 1 and 2)
4.	$\{A \wedge B\} \vdash B \wedge A$	(by conjE with 3)
5.	$\{\} \vdash A \wedge B \longrightarrow B \wedge A$	(by impI with 4)

Logic and Meta-Logic

Our logic gives us different ways to establish "X implies Y":

$$\{X\} \vdash Y \quad \{\} \vdash X \longrightarrow Y \quad \frac{\{\} \vdash Y}{\{\} \vdash X}$$

When one logic is embedded in another, we call the outer logic a meta-logic. If we were to discuss Spanish grammar, we would (probably) be using English as a meta-language. It is not uncommon to have chains of meta-meta-logics etc.

A formal logic L could be precisely defined in an outer meta-logic.

- so we can prove theorems about what L can prove

"Logic dictates the needs of the many outweigh the needs of the few."

What is a theorem prover?

An implementation of a formal logic on a computer.

Which logic?

- ➜ fully automated (propositional logic)
- ➜ automated, but not necessarily terminating (first order logic)
- ➜ with automation, but mainly interactive (higher order logic)

There are plenty of other (algorithmic) verification approaches:

- ➜ model checking, static analysis, ...
- ➜ See COMP3153: Algorithmic Verification, SENG2011, etc

Main theorem proving system for this course

Isabelle

➜ used at UNSW for research, teaching and proof engineering

`https://isabelle.in.tum.de/`

What is Isabelle?

A generic interactive proof assistant

- ➜ **generic:**
  not specialised to one particular logic
  (two large developments: HOL and ZF, will mainly use HOL)
- ➜ **interactive:**
  more than just yes/no, you can interactively guide the system
- ➜ **proof assistant:**
  helps to explore, find, and maintain proofs

If I prove it on the computer, it is correct, right?

If I prove it on the computer, it is correct, right?

No, because:

① hardware could be faulty
② operating system could be faulty
③ implementation runtime system could be faulty
④ compiler could be faulty
⑤ implementation could be
⑥ logic could be inconsistent
⑦ theorem could mean something else

If I prove it on the computer, it is correct, right?

No, but:
probability for

- ➜ OS and H/W issues reduced by using different systems
- ➜ runtime/compiler bugs reduced by using different compilers
- ➜ faulty implementation reduced by having the right prover architecture
- ➜ inconsistent logic reduced by implementing and analysing it
- ➜ wrong theorem reduced by expressive/intuitive logics

No guarantees, but assurance immensly higher than manual proof

If I prove it on the computer, it is correct, right?

Soundness architectures

careful implementation	PVS ACL2
LCF approach, small proof kernel	HOL4 Isabelle HOL-light
explicit proofs + proof checker	Coq Lean Twelf Isabelle HOL4 Agda

Isabelle's Meta Logic

$$\bigwedge \qquad\qquad \Longrightarrow \qquad\qquad \lambda$$

$\bigwedge$

Syntax: $\bigwedge x.\ F$ (F another meta logic formula)
in ASCII: `!!x. F`

➜ this is the meta-logic universal quantifier
➜ example and more later

$\Longrightarrow$

Syntax: $A \Longrightarrow B$ (A, B other meta logic formulae)
in ASCII: `A ==> B`

Binds to the right:

$$A \Longrightarrow B \Longrightarrow C \quad = \quad A \Longrightarrow (B \Longrightarrow C)$$

Abbreviation:

$$[\![A; B]\!] \Longrightarrow C \quad = \quad A \Longrightarrow B \Longrightarrow C$$

- ➜ read: A and B implies C
- ➜ used to write down rules, theorems, and proof states

Example: a theorem

mathematics: if $x < 0$ and $y < 0$, then $x + y < 0$

formal logic: $\vdash\ x < 0 \wedge y < 0 \longrightarrow x + y < 0$

variation: $x < 0; y < 0\ \vdash\ x + y < 0$

Isabelle: **lemma** "$x < 0 \wedge y < 0 \longrightarrow x + y < 0$"

variation: **lemma** "$[\![x < 0; y < 0]\!] \Longrightarrow x + y < 0$"

variation: **lemma**

assumes "$x < 0$" and "$y < 0$" shows "$x + y < 0$"

Example: a rule

logic:

$$\frac{X \quad Y}{X \wedge Y}$$

variation:

$$\frac{S \vdash X \quad S \vdash Y}{S \vdash X \wedge Y}$$

Isabelle:

$$[\![X; Y]\!] \Longrightarrow X \wedge Y$$

Example: a rule with nested implication

logic:

$$\frac{X \vee Y \quad \begin{array}{c} X \\ \vdots \\ Z \end{array} \quad \begin{array}{c} Y \\ \vdots \\ Z \end{array}}{Z}$$

variation:

$$\frac{S \cup \{X\} \vdash Z \quad S \cup \{Y\} \vdash Z}{S \cup \{X \vee Y\} \vdash Z}$$

Isabelle: $[\![X \vee Y; X \Longrightarrow Z; Y \Longrightarrow Z]\!] \Longrightarrow Z$

λ

Syntax: $\lambda x.\ F$ (F another meta logic formula)
in ASCII: `%x. F`

- ➜ lambda abstraction
- ➜ used to represent functions
- ➜ used to encode bound variables
- ➜ more about this soon

Enough Theory!

Getting started with Isabelle

System Architecture

Prover IDE (jEdit) – user interface

HOL, ZF – object-logics

Isabelle – generic, interactive theorem prover

Standard ML – logic implemented as ADT

User can access all layers!

System Requirements

➜ **Linux**, **Windows**, or **MacOS X (10.8 +)**

Premade packages for Linux, Mac, and Windows + info on:
`https://isabelle.in.tum.de/`

➜ We will use Isabelle 2024 in this iteration of COMP4161.

➜ The installer will fetch **PolyML**, **Java** and other dependencies itself. The install process is fairly smooth.

➜ Battery warning: Requires $\approx$ 2-3GB download, 5-10GB disk space, 5-10 minutes CPU time to set up.

Documentation

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

- ➜ Learning Isabelle
  - Concrete Semantics Book
  - Tutorial on Isabelle/HOL (LNCS 2283)
  - Tutorial on Isar
- ➜ Reference Manuals
  - Isabelle/Isar Reference Manual
  - Isabelle Reference Manual
  - Isabelle System Manual
- ➜ Reference Manuals for Object Logics

Ready for a Demo?

First: a word from our sponsor.

About us: UNSW Trustworthy Systems

TS (Trustworthy Systems) is a research group at UNSW.

- ➜ An alliance of systems developers and formal methods practitioners.
- ➜ A track record of research and real world impact in verified software.
- ➜ Biggest single achievement: formal verification of **seL4**.

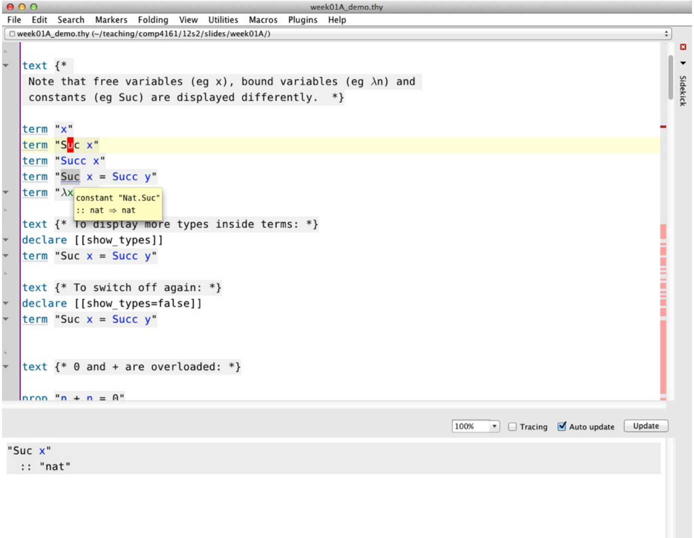
seL4: an OS microkernel with a strong security design

- ➜ Designed at UNSW.
- ➜ Implemented in $\approx 10\,000$ lines of low-level C code.
- ➜ Verified in over 1 million lines of Isabelle/HOL proofs.
  - ➜ Now maintained by **Proofcraft**.
- ➜ Used in critical systems, commercial & research, around the world.

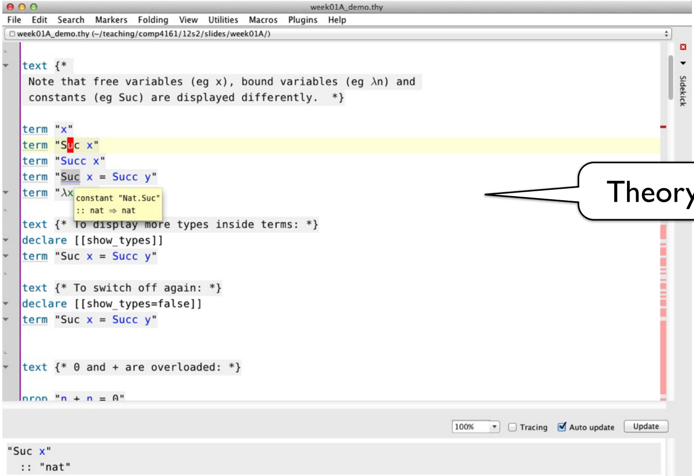
We are always embarking on exciting new projects. Talk to us!

DEMO

jEdit/PIDE


week01A_demo.thy
File Edit Search Markers Folding View Utilities Macros Plugins Help
week01A_demo.thy (~/teaching/comp4161/12s2/slides/week01A/)
text {*
Note that free variables (eg x), bound variables (eg λn) and
constants (eg Suc) are displayed differently. *}
term "x"
term "Suc x"
term "Succ x"
term "Suc x = Succ y"
term "λx
constant "Nat.Suc"
:: nat ⇒ nat
text {* To display more types inside terms: *}
declare [[show_types]]
term "Suc x = Succ y"
text {* To switch off again: *}
declare [[show_types=false]]
term "Suc x = Succ y"
text {* 0 and + are overloaded: *}
Sidekick
100%
Tracing
Auto update
Update
"Suc x"
:: "nat"

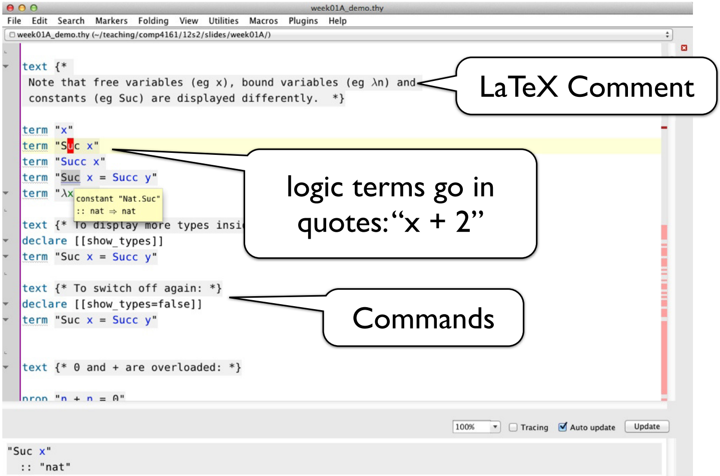
jEdit/PIDE



Theory File

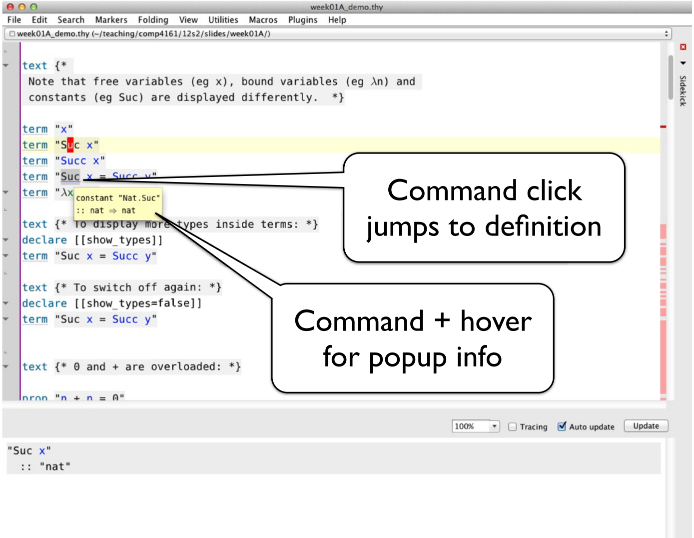
Isabelle Output

jEdit/PIDE



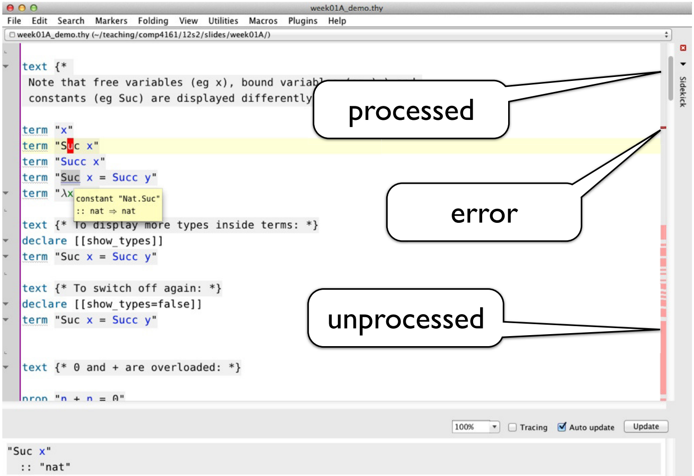
week01A_demo.thy
File Edit Search Markers Folding View Utilities Macros Plugins Help
week01A_demo.thy (~/teaching/comp4161/12s2/slides/week01A/)
text {*
Note that free variables (eg x), bound variables (eg λn) and
constants (eg Suc) are displayed differently. *}
term "x"
term "Suc x"
term "Succ x"
term "Suc x = Succ y"
term "λx
constant "Nat.Suc"
:: nat ⇒ nat
text {* To display more types insi
declare [[show_types]]
term "Suc x = Succ y"
text {* To switch off again: *}
declare [[show_types=false]]
term "Suc x = Succ y"
text {* 0 and + are overloaded: *}
100%
Tracing
Auto update
Update
"Suc x"
:: "nat"
LaTeX Comment
logic terms go in quotes: "x + 2"
Commands

jEdit/PIDE



week01A_demo.thy
File Edit Search Markers Folding View Utilities Macros Plugins Help
week01A_demo.thy (~/teaching/comp4161/12s2/slides/week01A/)
text {*
Note that free variables (eg x), bound variables (eg λn) and
constants (eg Suc) are displayed differently. *}
term "x"
term "Suc x"
term "Succ x"
term "Suc x = Succ y"
term "λx
constant "Nat.Suc"
:: nat ⇒ nat
text {* To display more types inside terms: *}
declare [[show_types]]
term "Suc x = Succ y"
text {* To switch off again: *}
declare [[show_types=false]]
term "Suc x = Succ y"
text {* 0 and + are overloaded: *}
prop "n + n = 0"
Sidekick
100%
Tracing
Auto update
Update
"Suc x"
:: "nat"
Command click jumps to definition
Command + hover for popup info

jEdit/PIDE



week01A_demo.thy
File Edit Search Markers Folding View Utilities Macros Plugins Help
week01A_demo.thy (~/teaching/comp4161/12s2/slides/week01A/)
text {*
Note that free variables (eg x), bound variab
constants (eg Suc) are displayed differently
term "x"
term "Suc x"
term "Succ x"
term "Suc x = Succ y"
term "λx
constant "Nat.Suc"
:: nat ⇒ nat
text {* To display more types inside terms: *}
declare [[show_types]]
term "Suc x = Succ y"
text {* To switch off again: *}
declare [[show_types=false]]
term "Suc x = Succ y"
text {* 0 and + are overloaded: *}
processed
error
unprocessed
Sidekick
100%
Tracing
Auto update
Update
"Suc x"
:: "nat"

Exercises

- Download and install Isabelle from `https://isabelle.in.tum.de/`
- Step through the demo files from the lecture web page
- Write your own theory file, look at some theorems in the library, try 'find_theorems'
- How many theorems can help you if you need to prove something containing the term "Suc(Suc x)"?
- What is the name of the theorem for associativity of addition of natural numbers in the library?

λ-Calculus

Content

- Foundations & Principles
  - Intro, Lambda calculus, natural deduction [1,2]
  - Higher Order Logic, Isar (part 1) [2,3[a]]
  - Term rewriting [3,4]

- Proof & Specification Techniques
  - Inductively defined sets, rule induction [4,5]
  - Datatype induction, primitive recursion [5,7]
  - General recursive functions, termination proofs [7]
  - Proof automation, Isar (part 2) [8[b]]
  - Hoare logic, proofs about programs, invariants [8,9]
  - C verification [9,10]
  - Practice, questions, exam prep [10[c]]

[a] a1 due; [b] a2 due; [c] a3 due

λ-calculus

Alonzo Church

- ➜ lived 1903–1995
- ➜ supervised people like Alan Turing, Stephen Kleene
- ➜ famous for Church-Turing thesis, lambda calculus,
  first undecidability results
- ➜ invented λ calculus in 1930's
- ➜ invented HOL

λ-calculus

- ➜ originally meant as foundation of mathematics
- ➜ important applications in theoretical computer science
- ➜ foundation of computability and functional programming
- ➜ one of the building blocks of HOL

untyped λ-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

Function Application

For applying arguments to functions

instead of	$f(a)$
write	$f\ a$

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 $(a + b) + 5$

That's it!

Now Formal

Syntax

Terms: $t \quad ::= \quad v \mid c \mid (t\ t) \mid (\lambda x.\ t)$

$v, x \in V, \quad c \in C, \quad V, C$ 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 λ

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

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

Intuition: replace parameter by argument
this is called β-reduction

Remember: $(\lambda x.\ t)\ a$ is evaluated (noted $\longrightarrow_\beta$) to
t where x is replaced by a

Example

$$
\begin{array}{l}
(\lambda x\ y.\ Suc\ x\ =\ y)\ 3 \equiv \\
(\lambda x.\ (\lambda y.\ Suc\ x\ =\ y))\ 3 \longrightarrow_\beta \\
(\lambda y.\ Suc\ 3\ =\ y)
\end{array}
$$

$$
\begin{array}{l}
(\lambda x\ y.\ f\ (y\ x))\ \ 5\ \ (\lambda x.\ x) \longrightarrow_\beta \\
(\lambda y.\ f\ (y\ 5))\ \ (\lambda x.\ x) \longrightarrow_\beta \\
f\ ((\lambda x.\ x)\ 5) \longrightarrow_\beta \\
f\ 5
\end{array}
$$

Defining Computation

β reduction:

$$
\begin{array}{ccccrcl}
 & & & & (\lambda x.\ s)\ t & \longrightarrow_\beta & s[x \leftarrow t] \\
s & \longrightarrow_\beta & s' & \Longrightarrow & (s\ t) & \longrightarrow_\beta & (s'\ t) \\
t & \longrightarrow_\beta & t' & \Longrightarrow & (s\ t) & \longrightarrow_\beta & (s\ t') \\
s & \longrightarrow_\beta & s' & \Longrightarrow & (\lambda x.\ s) & \longrightarrow_\beta & (\lambda x.\ s')
\end{array}
$$

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

Defining Substitution

Easy concept. Small problem: variable capture.
Example: $(\lambda x.\ x\ z)[z \leftarrow x]$

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

What do we want?

In $(\lambda y.\ y\ z)\ [z \leftarrow x] = (\lambda y.\ y\ x)$ 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:

$$
\begin{aligned}
FV\ (x) &= \{x\} \\
FV\ (c) &= \{\} \\
FV\ (s\ t) &= FV(s) \cup FV(t) \\
FV\ (\lambda x.\ t) &= FV(t) \setminus \{x\}
\end{aligned}
$$

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

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

The substitution example, $(\lambda x.\ x\ z)[z \leftarrow x]$, is problematic because the bound variable x is a free variable of the replacement term "x".

Substitution

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

Substitution Example

$$\begin{array}{rl} & (x \;\; (\lambda x.\, x) \;\; (\lambda y.\, z\ x))[x \leftarrow y] \\ = & (x[x \leftarrow y]) \;\; ((\lambda x.\, x)[x \leftarrow y]) \;\; ((\lambda y.\, z\ x)[x \leftarrow y]) \\ = & y\ (\lambda x.\, x)\ (\lambda y'.\, z\ y) \end{array}$$

α Conversion

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

α **conversion:**
$s =_\alpha t$ means $s = t$ up to renaming of bound variables.

Formally:

$$
\begin{array}{lllllll}
 & & & & (\lambda x.\ t) & \longrightarrow_\alpha & (\lambda y.\ t[x \leftarrow y]) \text{ if } y \notin FV(t) \\
s & \longrightarrow_\alpha & s' & \Longrightarrow & (s\ t) & \longrightarrow_\alpha & (s'\ t) \\
t & \longrightarrow_\alpha & t' & \Longrightarrow & (s\ t) & \longrightarrow_\alpha & (s\ t') \\
s & \longrightarrow_\alpha & s' & \Longrightarrow & (\lambda x.\ s) & \longrightarrow_\alpha & (\lambda x.\ s')
\end{array}
$$

$$s =_\alpha t \quad \text{iff} \quad s \longrightarrow_\alpha^* t$$

($\longrightarrow_\alpha^*$ = transitive, reflexive closure of $\longrightarrow_\alpha$ = multiple steps)

α Conversion

Equality in Isabelle is equality modulo α conversion:

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

Examples:

$$
\begin{array}{ll}
 & x\ (\lambda x\ y.\ x\ y) \\
=_\alpha & x\ (\lambda y\ x.\ y\ x) \\
=_\alpha & x\ (\lambda z\ y.\ z\ y) \\
\neq_\alpha & z\ (\lambda z\ y.\ z\ y) \\
\neq_\alpha & x\ (\lambda x\ x.\ x\ x)
\end{array}
$$

Back to β

We have defined β reduction: $\longrightarrow_\beta$
Some notation and concepts:

- β **conversion:** $s =_\beta t$ iff $\exists n.\ s \longrightarrow_\beta^* n \wedge t \longrightarrow_\beta^* n$
- t is **reducible** if there is an s such that $t \longrightarrow_\beta s$
- $(\lambda x.\ s)\ t$ 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 λ term have a normal form?

No!

Example:

$$
\begin{array}{l}
(\lambda x.\ x\ x)\ (\lambda x.\ x\ x) \ \longrightarrow_\beta \\
(\lambda x.\ x\ x)\ (\lambda x.\ x\ x) \ \longrightarrow_\beta \\
(\lambda x.\ x\ x)\ (\lambda x.\ x\ x) \ \longrightarrow_\beta \ \ldots
\end{array}
$$

(but: $(\lambda x\ y.\ y)\ ((\lambda x.\ x\ x)\ (\lambda x.\ x\ x)) \ \longrightarrow_\beta \ \lambda y.\ y$)

λ calculus is not terminating

β reduction is confluent

Confluence: $s \longrightarrow_\beta^* x \wedge s \longrightarrow_\beta^* y \Longrightarrow \exists t.\ x \longrightarrow_\beta^* t \wedge y \longrightarrow_\beta^* t$

$$\begin{tikzcd}
 & s \arrow[dl, "*"'] \arrow[dr, "*"] & \\
x \arrow[dr, dashed, "*"'] & & y \arrow[dl, dashed, "*"] \\
 & t &
\end{tikzcd}$$

Order of reduction does not matter for result
Normal forms in λ calculus are unique

β reduction is confluent

Example:

$(\lambda x\ y.\ y)\ ((\lambda x.\ x\ x)\ a) \longrightarrow_\beta (\lambda x\ y.\ y)\ (a\ a) \longrightarrow_\beta \lambda y.\ y$

$(\lambda x\ y.\ y)\ ((\lambda x.\ x\ x)\ a) \longrightarrow_\beta \lambda y.\ y$

η Conversion

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

Definition:

$$\begin{array}{llllrll}
 & & & & (\lambda x.\ t\ x) & \longrightarrow_\eta & t \qquad\qquad \text{if } x \notin FV(t) \\
s & \longrightarrow_\eta & s' & \Longrightarrow & (s\ t) & \longrightarrow_\eta & (s'\ t) \\
t & \longrightarrow_\eta & t' & \Longrightarrow & (s\ t) & \longrightarrow_\eta & (s\ t') \\
s & \longrightarrow_\eta & s' & \Longrightarrow & (\lambda x.\ s) & \longrightarrow_\eta & (\lambda x.\ s')
\end{array}$$

$$s =_\eta t \quad \text{iff} \quad \exists n.\ s \longrightarrow_\eta^* n \wedge t \longrightarrow_\eta^* n$$

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

- ➜ η reduction is confluent and terminating.
- ➜ $\longrightarrow_{\beta\eta}$ is confluent.
  $\longrightarrow_{\beta\eta}$ means $\longrightarrow_\beta$ and $\longrightarrow_\eta$ steps are both allowed.
- ➜ **Equality in Isabelle is also modulo η conversion.**

In fact ...

Equality in Isabelle is modulo α, β, and η conversion.

We will see later why that is possible.

Isabelle Demo

So, what can you do with λ calculus?

λ calculus is very expressive, you can encode:

- ➜ logic, set theory
- ➜ turing machines, functional programs, etc.

Examples:

$$\begin{array}{lll} \texttt{true} & \equiv \lambda x\ y.\ x & \qquad \texttt{if true}\ x\ y \longrightarrow_\beta^* x \\ \texttt{false} & \equiv \lambda x\ y.\ y & \qquad \texttt{if false}\ x\ y \longrightarrow_\beta^* y \\ \texttt{if} & \equiv \lambda z\ x\ y.\ z\ x\ y & \end{array}$$

Now, `not`, `and`, `or`, etc is easy:

$$\begin{array}{ll} \texttt{not} & \equiv \lambda x.\ \texttt{if}\ x\ \texttt{false true} \\ \texttt{and} & \equiv \lambda x\ y.\ \texttt{if}\ x\ y\ \texttt{false} \\ \texttt{or} & \equiv \lambda x\ y.\ \texttt{if}\ x\ \texttt{true}\ y \end{array}$$

More Examples

Encoding natural numbers (Church Numerals)

$$
\begin{array}{ll}
0 & \equiv \lambda f\ x.\ x \\
1 & \equiv \lambda f\ x.\ f\ x \\
2 & \equiv \lambda f\ x.\ f\ (f\ x) \\
3 & \equiv \lambda f\ x.\ f\ (f\ (f\ x)) \\
\ldots &
\end{array}
$$

Numeral n takes arguments f and x, applies f n-times to x.

$$
\begin{array}{ll}
\texttt{iszero} & \equiv \lambda n.\ n\ (\lambda x.\ \texttt{false})\ \texttt{true} \\
\texttt{succ} & \equiv \lambda n\ f\ x.\ f\ (n\ f\ x) \\
\texttt{add} & \equiv \lambda m\ n.\ \lambda f\ x.\ m\ f\ (n\ f\ x)
\end{array}
$$

Fix Points

$$(\lambda x\, f.\, f\, (x\, x\, f))\ \ (\lambda x\, f.\, f\, (x\, x\, f))\ \ t \longrightarrow_\beta$$
$$(\lambda f.\, f\, ((\lambda x\, f.\, f\, (x\, x\, f))\ \ (\lambda x\, f.\, f\, (x\, x\, f))\ f))\ \ t \longrightarrow_\beta$$
$$t\ \ ((\lambda x\, f.\, f\, (x\, x\, f))\ (\lambda x\, f.\, f\, (x\, x\, f))\ t)$$

$$\mu = (\lambda x\, f.\, f\, (x\, x\, f))\ (\lambda xf.\, f\, (x\, x\, f))$$
$$\mu\ t \longrightarrow_\beta t\ (\mu\ t) \longrightarrow_\beta t\ (t\ (\mu\ t)) \longrightarrow_\beta t\ (t\ (t\ (\mu\ t))) \longrightarrow_\beta \ldots$$

$(\lambda xf.\, f\, (x\, x\, f))\ (\lambda xf.\, f\, (x\, x\, f))$ is Turing's fix point operator

Nice, but ...

As a mathematical foundation, λ does not work. **It resulted in an inconsistent logic.**

- ➜ **Frege** (Predicate Logic, $\sim$ 1879):
  allows arbitrary quantification over predicates
- ➜ **Russell** (1901): Paradox $R \equiv \{X | X \notin X\}$
- ➜ **Whitehead & Russell** (Principia Mathematica, 1910-1913):
  Fix the problem
- ➜ **Church** (1930): λ calculus as logic, $\texttt{true}$, $\texttt{false}$, $\wedge$, $\ldots$ as λ terms

Problem:

with	$\{x \mid P\ x\} \equiv \lambda x.\ P\ x \qquad x \in M \equiv M\ x$
you can write	$R \equiv \lambda x.\ \texttt{not}\ (x\ x)$
and get	$(R\ R) =_\beta \texttt{not}\ (R\ R)$
because	$(R\ R) = (\lambda x.\ \texttt{not}\ (x\ x))\ R \longrightarrow_\beta \texttt{not}\ (R\ R)$

We have learned so far...

- ➜ λ calculus syntax
- ➜ free variables, substitution
- ➜ β reduction
- ➜ α and η conversion
- ➜ β reduction is confluent
- ➜ λ calculus is very expressive (turing complete)
- ➜ λ calculus results in an inconsistent logic