COMP 4161

Data61 Advanced Course

Advanced Topics in Software Verification

Gerwin Kein,
Johannes Åman Pohjola,
Christine Rizkallah
Miki Tanaka,
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;

http://googleresearch.blogspot.com/2006/06-extra-extra-read-all-about-it-nearly.html
Organisatorials

When
Tue 11:00 – 13:00
Wed 16:00 – 18:00

http://www.cse.unsw.edu.au/~cs4161/
About us

The trustworthy systems verification team

- Functional correctness and security of the seL4 microkernel
  Security ↔ Isabelle/HOL model ↔ Haskell model ↔ C code ↔ Binary
- 10 000 LOC / 500 000 lines of proof; about 25 person years of effort
- Cogent code/proof co-generation; CakeML verified compiler; etc.

Open Source

http://sel4.systems
https://cakeml.org

We are always embarking on exciting new projects.
We offer

- summer student scholarship projects
- honours and PhD theses
- research assistant and verification engineer positions
What you will learn

- how to use a theorem prover
- background, how it works
- how to prove and specify
- how to reason about programs

Health Warning

Theorem Proving is addictive
Prerequisites

This is an advanced course. It assumes knowledge in

→ Functional programming
→ First-order formal logic

The following program should make sense to you:

\[
\begin{align*}
\text{map } f \; [] &= [] \\
\text{map } f \; (x:xs) &= f \; x : \text{map } f \; xs
\end{align*}
\]

You should be able to read and understand this formula:

\[
\exists x. \ (P(x) \rightarrow \forall x. \ P(x))
\]
Content — Using Theorem Provers

Rough timeline

→ 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, datatype induction, primitive recursion [4,5]
  • General recursive functions, termination proofs [7^b]
  • Proof automation, Hoare logic, proofs about programs, invariants [8]
  • C verification [9,10]
  • Practice, questions, examp prep [10^c]

^a a1 due; ^b a2 due; ^c a3 due
What you should do to have a chance at succeeding

→ attend lectures
→ try Isabelle early
→ redo all the demos alone
→ try the exercises/homework we give, when we do give some

→ DO NOT CHEAT
  ● 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

<https://student.unsw.edu.au/plagiarism>
Credits

some material (in using-theorem-provers part) shamelessly stolen from

Tobias Nipkow, Larry Paulson, Markus Wenzel

David Basin, Burkhardt Wolff

Don’t blame them, errors are ours
What is a formal proof?

A derivation in a formal calculus

Example: \( A \land B \rightarrow B \land A \) derivable in the following system

Rules:
- \( X \in S \rightarrow S \vdash X \) (assumption)
- \( S \vdash X \rightarrow Y \) (impl)
- \( S \vdash X \land Y \) (conj)
- \( S \cup \{ X, Y \} \vdash Z \) (conjE)

Proof:
1. \( \{ A, B \} \vdash B \) (by assumption)
2. \( \{ A, B \} \vdash A \) (by assumption)
3. \( \{ A, B \} \vdash B \land A \) (by conj with 1 and 2)
4. \( \{ A \land B \} \vdash B \land A \) (by conjE with 3)
5. \( \{ \} \vdash A \land B \rightarrow B \land A \) (by impl with 4)
What is a theorem prover?

Implementation of a formal logic on a computer.

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

- based on rules and axioms
- can deliver proofs

There are other (algorithmic) verification tools:

- model checking, static analysis, ...
- usually do not deliver proofs
- See COMP3153: Algorithmic Verification
Why theorem proving?

- Analysing systems/programs thoroughly
- Finding design and specification errors early
- High assurance (mathematical, machine checked proof)
- it’s not always easy
- it’s fun
Main theorem proving system for this course

Isabelle

→ used here for applications, learning how to prove
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 faulty
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 immensely higher than manual proof
If I prove it on the computer, it is correct, right?

<table>
<thead>
<tr>
<th>Soundness architectures</th>
<th>PVS</th>
</tr>
</thead>
<tbody>
<tr>
<td>careful implementation</td>
<td></td>
</tr>
<tr>
<td>LCF approach, small proof kernel</td>
<td>HOL4, Isabelle</td>
</tr>
<tr>
<td>explicit proofs + proof checker</td>
<td>Coq, Twelf, Isabelle, HOL4</td>
</tr>
</tbody>
</table>
Meta Logic

Meta language:
The language used to talk about another language.

Examples:
English in a Spanish class, English in an English class

Meta logic:
The logic used to formalize another logic

Example:
Mathematics used to formalize derivations in formal logic
Meta Logic – Example

Syntax:
Formulae: \[ F ::= V \mid F \rightarrow F \mid F \land F \mid False \]
\[ V ::= [A - Z] \]

Derivable: \[ S \vdash X \quad X \text{ a formula, } S \text{ a set of formulae} \]

Logic / Meta logic

\[
\begin{align*}
X & \in S \\
\therefore S \vdash X
\end{align*}
\]

Logic

\[
\begin{align*}
S \cup \{X\} & \vdash Y \\
\therefore S \vdash X \rightarrow Y
\end{align*}
\]

Logic

\[
\begin{align*}
S \vdash X & \quad S \vdash Y \\
\therefore S \vdash X \land Y
\end{align*}
\]

Logic

\[
\begin{align*}
S \cup \{X, Y\} & \vdash Z \\
S \cup \{X \land Y\} & \vdash Z
\end{align*}
\]

Logic
Isabelle’s Meta Logic

∧  ⇒  λ
**Syntax:** \( \forall x. F \)  
(F another meta level formula)  
in ASCII: \( ! ! x. F \)

- universal quantifier on the meta level  
- used to denote parameters  
- example and more later
Syntax: \[ A \implies B \]  
(A, B other meta level formulae)

in ASCII: \[ A \implies B \]

Binds to the right:

\[ A \implies B \implies C \] = \[ A \implies (B \implies C) \]

Abbreviation:

\[ [A; B] \implies C \] = \[ A \implies B \implies C \]

→ 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: $x < 0 \land y < 0 \implies x + y < 0$

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

Isabelle: lemma “$x < 0 \land y < 0 \implies x + y < 0$”

variation: lemma “[x < 0; y < 0] \implies x + y < 0”

variation: lemma assumes “$x < 0$” and “$y < 0$” shows “$x + y < 0$”
Example: a rule

logic: 
\[
\begin{array}{c|c}
X & Y \\
\hline
X & \land & Y
\end{array}
\]

variation: 
\[
\begin{array}{c}
S \vdash X \\
S \vdash Y
\end{array} \\
\hline
S \vdash X \land Y
\]

Isabelle: 
\[
[X; Y] \implies X \land Y
\]
Example: a rule with nested implication

\[
\begin{array}{cccc}
\text{X} & \text{Y} & \text{Z} & \text{Z} \\
\hline
\text{X} & \text{Y} & \text{Z} \\
\end{array}
\]

logic:

\[
\frac{X \lor Y}{Z} 
\]

variation:

\[
\frac{S \cup \{X\} \vdash Z \\ S \cup \{Y\} \vdash Z}{S \cup \{X \lor Y\} \vdash Z} 
\]

Isabelle:

\[
\llbracket X \lor Y; X \Rightarrow Z; Y \Rightarrow Z \rrbracket \Rightarrow Z 
\]
Syntax: $\lambda x. F$  
(F another meta level formula)

in ASCII: %x. F

- lambda abstraction
- used for functions in object logics
- used to encode bound variables in object logics
- more about this in the next lecture
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 +)
→ Standard ML (PolyML implementation)
→ Java (for jEdit)

Premade packages for Linux, Mac, and Windows + info on: http://mirror.cse.unsw.edu.au/pub/isabelle/
Documentation

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

- Learning Isabelle
  - Concrete Semantics Book
  - 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
Demo
text {*
    Note that free variables (eg x), bound variables (eg \( \lambda n \)) and
    constants (eg Succ) are displayed differently. *}

term "x"
term "Succ x"
term "Succ x"
term "Succ x = Succ y"
term "\( \lambda x \) constant "Nat.Suc" :: nat -> nat"
text {* To display more types inside terms: *}
declare [[show_types]]
term "Succ x = Succ y"

text {* To switch off again: *}
declare [[show_types=false]]
term "Succ x = Succ y"

text {* 0 and + are overloaded: *}
prop "x + x = x"

"Succ x"
:: "nat"
Theory File

Isabelle Output
LaTeX Comment

logic terms go in quotes: “x + 2”

Commands
Command click jumps to definition

Command + hover for popup info
Exercises

→ Download and install Isabelle from http://mirror.cse.unsw.edu.au/pub/isabelle/
→ 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, datatype induction, primitive recursion [4,5]
  - General recursive functions, termination proofs [7^b]
  - Proof automation, Hoare logic, proofs about programs, invariants [8]
  - C verification [9,10]
  - Practice, questions, examp prep [10^c]

^a1 due; ^b2 due; ^c3 due
\textbf{\textit{\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 \textit{\lambda} calculus in 1930’s

\textbf{\textit{\lambda}}-calculus
- originally meant as foundation of mathematics
- important applications in theoretical computer science
- foundation of computability and functional programming
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
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: \[ t ::= v \mid c \mid (t \ t) \mid (\lambda x. \ t) \]
\[ v, x \in V, \quad c \in C, \quad V, C \text{ sets of names} \]

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

- leave out parentheses where possible
- list variables instead of multiple 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: \(x y z = (x y) z \not= x (y z)\)
- abstraction binds to the right: \(\lambda x. x y = \lambda x. (x y) \not= (\lambda x. x) y\)
- 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 $\beta$-reduction

Example

$$(\lambda x \ y. \ f \ (y \ x)) \ 5 \ (\lambda x. \ x) \to^\beta$$

$$(\lambda y. \ f \ (y \ 5)) \ (\lambda x. \ x) \to^\beta$$

$$f \ ((\lambda x. \ x) \ 5) \to^\beta$$

$$f \ 5$$
Defining Computation

\[ \beta \text{ reduction:} \]

\[
\begin{align*}
(\lambda x. s) t & \rightarrow_\beta s[x \leftarrow t] \\
 s & \rightarrow_\beta s' \quad \Rightarrow \quad (s t) & \rightarrow_\beta (s' t) \\
 t & \rightarrow_\beta t' \quad \Rightarrow \quad (s t) & \rightarrow_\beta (s t') \\
 s & \rightarrow_\beta s' \quad \Rightarrow \quad (\lambda x. s) & \rightarrow_\beta (\lambda x. s')
\end{align*}
\]

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?

\[\text{In } (\lambda y. y z) [z \leftarrow x] = (\lambda y. y x)\] 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:

\[
\begin{align*}
FV(x) &= \{x\} \\
FV(c) &= \{\} \\
FV(s t) &= FV(s) \cup FV(t) \\
FV(\lambda x. t) &= FV(t) \setminus \{x\}
\end{align*}
\]

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

\[ x [x \leftarrow t] = t \]
\[ y [x \leftarrow t] = y \] if \( x \neq y \)
\[ c [x \leftarrow t] = c \]

\[ (s_1 s_2) [x \leftarrow t] = (s_1[x \leftarrow t] s_2[x \leftarrow t]) \]

\[ (\lambda x. s) [x \leftarrow t] = (\lambda x. s) \]
\[ (\lambda y. s) [x \leftarrow t] = (\lambda y. s[x \leftarrow t]) \] if \( x \neq y \) and \( y \notin FV(t) \)
\[ (\lambda y. s) [x \leftarrow t] = (\lambda z. s[y \leftarrow z][x \leftarrow t]) \] if \( x \neq y \) and \( z \notin FV(t) \cup FV(s) \)
Substitution Example

\[
(x \ (\lambda x. \ x) \ (\lambda y. \ z \ x))[x \leftarrow y]
= \ (x[x \leftarrow y]) \ ((\lambda x. \ x)[x \leftarrow y]) \ ((\lambda y. \ z \ x)[x \leftarrow y])
= \ y \ (\lambda x. \ x) \ (\lambda y'. \ z \ y)
\]
\[\alpha\] Conversion

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

**\[\alpha\] conversion:**
\[s =_{\alpha} t\] means \[s = t\] up to renaming of bound variables.

**Formally:**
\[
\begin{align*}
(\lambda x.\ t) & \rightarrow_{\alpha} (\lambda y.\ t[x \leftarrow y]) & \text{if } y \notin FV(t) \\
s & \rightarrow_{\alpha} s' & \Rightarrow & (s\ t) & \rightarrow_{\alpha} (s'\ t) \\
t & \rightarrow_{\alpha} t' & \Rightarrow & (s\ t) & \rightarrow_{\alpha} (s\ t') \\
s & \rightarrow_{\alpha} s' & \Rightarrow & (\lambda x.\ s) & \rightarrow_{\alpha} (\lambda x.\ s')
\end{align*}
\]

\[s =_{\alpha} t\] iff \[s \rightarrow^{*}_{\alpha} t\]
\[\rightarrow^{*}_{\alpha} = \text{transitive, reflexive closure of } \rightarrow_{\alpha} = \text{multiple steps}\]
Equality in Isabelle is equality modulo $\alpha$ conversion:

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

Examples:

$x \ (\lambda x \ y. \ x \ y)$

$=_{\alpha} \ x \ (\lambda y \ x. \ y \ x)$

$=_{\alpha} \ x \ (\lambda z \ y. \ z \ y)$

$\not=_{\alpha} \ z \ (\lambda z \ y. \ z \ y)$

$\not=_{\alpha} \ x \ (\lambda x \ x. \ x \ x)$
We have defined $\beta$ reduction: $\rightarrow_\beta$

Some notation and concepts:

- $\beta$ conversion: $s =_\beta t$ iff $\exists n. s \rightarrow_\beta^* n \land t \rightarrow_\beta^* n$
- $t$ is reducible if there is an $s$ such that $t \rightarrow_\beta s$
- $(\lambda x. s) t$ 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 $\lambda$ 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:  \[ s \xrightarrow{\beta} x \wedge s \xrightarrow{\beta} y \implies \exists t. \ x \xrightarrow{\beta} t \wedge y \xrightarrow{\beta} t \]

Order of reduction does not matter for result
Normal forms in \( \lambda \) calculus are unique
\( \beta \) reduction is confluent

Example:

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

\[
(\lambda x \ y. \ y) \ ((\lambda x. \ x \ x) \ a) \rightarrow^\beta \lambda y. \ y
\]
Another case of trivially equal functions: $t = (\lambda x. t x)$

Definition:

\[
\begin{align*}
(\lambda x. t x) & \to^\eta t & \text{if } x \notin \text{FV}(t) \\
(\lambda x. s) & \to^\eta (\lambda x. s') & \\
(s t) & \to^\eta (s' t) & \\
(s t) & \to^\eta (s t') & \\
(\lambda x. s) & \to^\eta (\lambda x. s') & \\
\end{align*}
\]

$s =^\eta t$ iff $\exists n. s \to^\ast n \land t \to^\ast n$

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

- $\eta$ reduction is confluent and terminating.
- $\to^\eta$ is confluent.
- $\to^\beta$ means $\to^\eta$ and $\to^\beta$ 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.
Isabelle Demo
So, what can you do with $\lambda$ calculus?

$\lambda$ calculus is very expressive, you can encode:

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

Examples:

- $\text{true} \equiv \lambda x \ y. \ x$  
  $\text{if} \ \text{true} \ x \ y \rightarrow^* \ x$

- $\text{false} \equiv \lambda x \ y. \ y$  
  $\text{if} \ \text{false} \ x \ y \rightarrow^* \ y$

- $\text{if} \ \equiv \lambda z \ x \ y. \ z \ x \ y$

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

- $\text{not} \equiv \lambda x. \ \text{if} \ x \ \text{false} \ \text{true}$
- $\text{and} \equiv \lambda x \ y. \ \text{if} \ x \ y \ \text{false}$
- $\text{or} \ \equiv \lambda x \ y. \ \text{if} \ x \ \text{true} \ y$
More Examples

Encoding natural numbers (Church Numerals)

\[ 0 \equiv \lambda f \ x. \ x \]
\[ 1 \equiv \lambda f \ x. \ f \ x \]
\[ 2 \equiv \lambda f \ x. \ f \ (f \ x) \]
\[ 3 \equiv \lambda f \ x. \ f \ (f \ (f \ x)) \]
\[ \ldots \]

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

\[ \text{iszero} \equiv \lambda n. \ n \ (\lambda x. \ false) \ true \]
\[ \text{succ} \equiv \lambda n \ f \ x. \ f \ (n \ f \ x) \]
\[ \text{add} \equiv \lambda m \ n. \ \lambda f \ x. \ m \ f \ (n \ f \ x) \]
Fix Points

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

\[\mu = (\lambda x \ f. \ f \ (x \ x \ f)) \ (\lambda x f. \ f \ (x \ x \ f))\]
\[\mu \ t \rightarrow_{\beta} t \ (\mu \ t) \rightarrow_{\beta} t \ (t \ (\mu \ t)) \rightarrow_{\beta} t \ (t \ (t \ (\mu \ t))) \rightarrow_{\beta} \ldots\]

\[(\lambda x f. \ f \ (x \ x \ f)) \ (\lambda x f. \ f \ (x \ x \ f)) \text{ is Turing's fix point operator}\]
As a mathematical foundation, $\lambda$ does not work. It resulted in an inconsistent logic.

- **Frege** (Predicate Logic, $\sim$ 1879): allows arbitrary quantification over predicates
- **Russell** (1901): Paradox $R \equiv \{X | X \not\in X\}$
- **Whitehead & Russell** (Principia Mathematica, 1910-1913): Fix the problem
- **Church** (1930): $\lambda$ calculus as logic, true, false, $\land$, ... as $\lambda$ terms

**Problem:**

with \( \{x | P \ x\} \equiv \lambda x. \ P \ x \quad x \in M \equiv M \ x \)
you can write \( R \equiv \lambda x. \ \text{not} \ (x \ x) \)
and get \( (R \ R) =_{\beta} \text{not} (R \ R) \)
because \( (R \ R) = (\lambda x. \ \text{not} \ (x \ x)) \ R \longrightarrow_{\beta} \text{not} (R \ R) \)
We have learned so far...

- $\lambda$ calculus syntax
- free variables, substitution
- $\beta$ reduction
- $\alpha$ and $\eta$ conversion
- $\beta$ reduction is confluent
- $\lambda$ calculus is very expressive (turing complete)
- $\lambda$ calculus results in an inconsistent logic