

COMP4161: Advanced Topics in Software Verification



Gerwin Klein, June Andronick, Christine Rizkallah, Miki Tanaka  
S2/2018

[data61.csiro.au](http://data61.csiro.au)



# Exercises from last time



- 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 " $\text{Suc}(\text{Suc } x)$ "?
- What is the name of the theorem for associativity of addition of natural numbers in the library?

# Content



- Intro & motivation, getting started [1]
  
- Foundations & Principles
  - Lambda Calculus, natural deduction [1,2]
  - Higher Order Logic [3<sup>a</sup>]
  - Term rewriting [4]
  
- Proof & Specification Techniques
  - Inductively defined sets, rule induction [5]
  - Datatypes, recursion, induction [6, 7]
  - Hoare logic, proofs about programs, invariants [8<sup>b</sup>, 9]
  - (mid-semester break)
  - C verification [10]
  - CakeML, Isar [11<sup>c</sup>]
  - Concurrency [12]

---

<sup>a</sup>a1 due; <sup>b</sup>a2 due; <sup>c</sup>a3 due

# $\lambda$ -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



## $\lambda$ -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  $(a + b) + 5$



DATA  
61



**That's it!**



DATA  
61



# Now Formal



# Syntax



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

$v, x \in V, \quad c \in C, \quad V, C$  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  $\lambda y x. x y$

## 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  $\beta$ -reduction

## Example

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

# Defining Computation



$\beta$  reduction:

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

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:

$$FV(x) = \{x\}$$

$$FV(c) = \{\}$$

$$FV(st) = FV(s) \cup FV(t)$$

$$FV(\lambda x. t) = FV(t) \setminus \{x\}$$

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

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



$$\begin{aligned}x [x \leftarrow t] &= t \\y [x \leftarrow t] &= y \\c [x \leftarrow t] &= c\end{aligned}\quad \text{if } x \neq y$$

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

$$(\lambda x. s) [x \leftarrow t] = (\lambda x. s)$$

$$(\lambda y. s) [x \leftarrow t] = (\lambda y. s[x \leftarrow t]) \quad \text{if } x \neq y \text{ and } y \notin FV(t)$$

$$\begin{aligned}(\lambda y. s) [x \leftarrow t] &= (\lambda z. s[y \leftarrow z][x \leftarrow t]) \quad \text{if } x \neq y \\ &\text{and } z \notin FV(t) \cup FV(s)\end{aligned}$$



# Substitution Example



$$\begin{aligned} & (x \ (\lambda x. x) \ (\lambda y. z \ x))[x \leftarrow y] \\ = & (x[x \leftarrow y]) \ ((\lambda x. x)[x \leftarrow y]) \ ((\lambda y. z \ x)[x \leftarrow y]) \\ = & y \ (\lambda x. x) \ (\lambda y'. z \ y) \end{aligned}$$

# $\alpha$ 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:**

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

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

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

# $\alpha$ Conversion



**Equality in Isabelle is equality modulo  $\alpha$  conversion:**

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

**Examples:**

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

# Back to $\beta$



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

Some notation and concepts:

- $\beta$  **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 $\lambda$ term have a normal form?



**No!**

**Example:**

$$\begin{aligned}(\lambda x. x x) (\lambda x. x x) &\longrightarrow_{\beta} \\(\lambda x. x x) (\lambda x. x x) &\longrightarrow_{\beta} \\(\lambda x. x x) (\lambda x. x x) &\longrightarrow_{\beta} \dots\end{aligned}$$

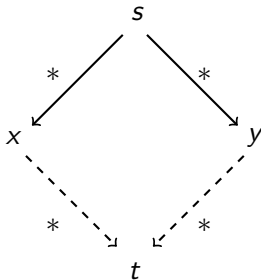
$$\text{(but: } (\lambda x y. y) ((\lambda x. x x) (\lambda x. x x)) \longrightarrow_{\beta} \lambda y. y)$$

**$\lambda$  calculus is not terminating**

# $\beta$ reduction is confluent



**Confluence:**  $s \longrightarrow_{\beta}^* x \wedge s \longrightarrow_{\beta}^* y \implies \exists t. x \longrightarrow_{\beta}^* t \wedge y \longrightarrow_{\beta}^* t$



**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:

$$\begin{aligned} s &\longrightarrow_{\eta} s' \implies (\lambda x. t x) \longrightarrow_{\eta} t && \text{if } x \notin FV(t) \\ t &\longrightarrow_{\eta} t' \implies (s t) \longrightarrow_{\eta} (s' t) \\ s &\longrightarrow_{\eta} s' \implies (s t) \longrightarrow_{\eta} (s t') \\ s &\longrightarrow_{\eta} s' \implies (\lambda x. s) \longrightarrow_{\eta} (\lambda x. s') \end{aligned}$$

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

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

- $\eta$  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  $\eta$  conversion.**



# In fact ...



**Equality in Isabelle is modulo  $\alpha$ ,  $\beta$ , and  $\eta$  conversion.**

We will see later why that is possible.

# 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:

`true`  $\equiv \lambda x y. x$

`if true`  $x y \longrightarrow_{\beta}^* x$

`false`  $\equiv \lambda x y. y$

`if false`  $x y \longrightarrow_{\beta}^* y$

`if`  $\equiv \lambda z x y. z x y$

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

`not`  $\equiv \lambda x. \text{if } x \text{ false true}$

`and`  $\equiv \lambda x y. \text{if } x y \text{ false}$

`or`  $\equiv \lambda x y. \text{if } x \text{ true } y$

# More Examples



## Encoding natural numbers (Church Numerals)

$$\begin{aligned}0 &\equiv \lambda f x. x \\1 &\equiv \lambda f x. f x \\2 &\equiv \lambda f x. f (f x) \\3 &\equiv \lambda f x. f (f (f x)) \\&\dots\end{aligned}$$

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

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

# Fix Points



$$\begin{aligned} & (\lambda x f. f (x x f)) (\lambda x f. f (x x f)) t \longrightarrow_{\beta} \\ & (\lambda f. f ((\lambda x f. f (x x f)) (\lambda x f. f (x x f)) f)) t \longrightarrow_{\beta} \\ & t ((\lambda x f. f (x x f)) (\lambda x f. f (x x f)) t) \end{aligned}$$

$$\begin{aligned} \mu &= (\lambda x f. f (x x f)) (\lambda x f. f (x x f)) \\ \mu t &\longrightarrow_{\beta} t (\mu t) \longrightarrow_{\beta} t (t (\mu t)) \longrightarrow_{\beta} t (t (t (\mu t))) \longrightarrow_{\beta} \dots \end{aligned}$$

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

# Nice, but ...



As a mathematical foundation,  $\lambda$  does not work. **It is inconsistent.**

- **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):  $\lambda$  calculus as logic, true, false,  $\wedge$ , ... as  $\lambda$  terms

## Problem:

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

A background pattern of white dashed lines forming a grid of hexagons on a teal background.

DATA  
61



# Isabelle Demo

# 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 is inconsistent