

CS 162 Programming languages

Lecture 6: λ -calculus II

Yu Feng
Winter 2022

Design a programming language

- Syntax: what do programs look like?
 - Grammar: what programs are we allowed to write?
- Semantics: what do programs mean?
 - Operational semantics: how do programs execute step-by-step?

Syntax: what programs look like

$$e ::= x$$
$$| \lambda x. e$$
$$| e_1 e_2$$

$\backslash x \rightarrow e$ (Haskell)

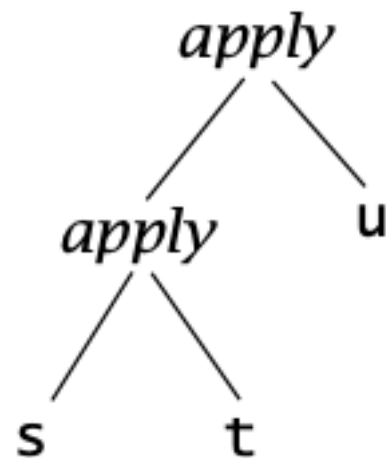
fun $x \rightarrow e$ (OCaml)

lambda $x. e$ ($\lambda+$)

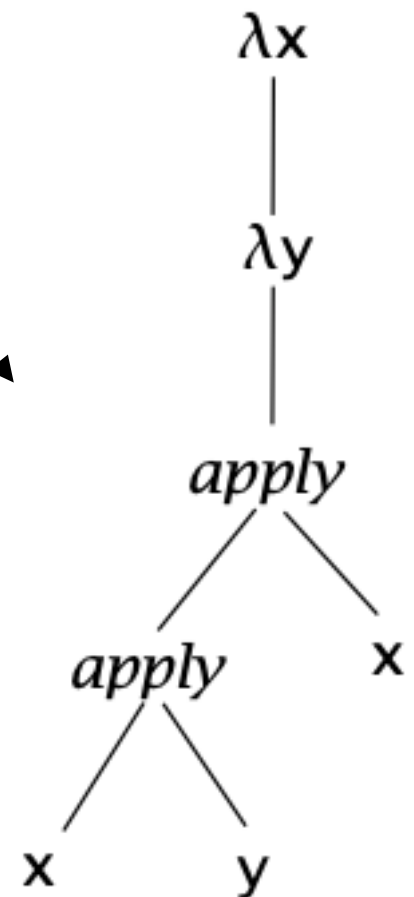
- Programs are expressions e (also called λ -terms) of one of three kinds:
 - Variable x, y, z
 - Abstraction (i.e. nameless function definition)
 - $\lambda x. e$
 - x is the formal parameter, e is the function body
 - Application (i.e. function call)
 - $e_1 e_2$
 - e_1 is the function, e_2 is the argument

Precedence

$$s \ t \ u = (s \ t) \ u$$



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



Application associates to the **left**

bodies of abstractions are as far to the **right** as possible

— *Types and programming languages*

Semantics: variable scope

The part of a program where a variable is visible

In the expression $\lambda x. e$

- x is the newly introduced variable
- e is the scope of x
- any occurrence of x in $\lambda x. e$ is bound (by the binder λx)

$\lambda x. x$

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

x is bounded

$x y$

$\lambda y. x y$

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

x is free

An occurrence of x in e is **free** if it's *not bound* by an enclosing abstraction

Semantics: free variables

An variable x is free in e if there exists a free occurrence of x in e

We use “FV” to represent the set of all free variables in a term:

$$FV(x) = x$$

$$FV(\lambda x. e) = FV(e) \setminus x$$

$$FV(e_1 e_2) = FV(e_1) \cup FV(e_2)$$

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

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

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

If e has no free variables it is said to be closed, or combinators

Semantics: what programs mean

- How do I execute a λ -term?
- “Execute”: rewrite step-by-step following simple rules, until no more rules apply

$e ::= x$
| $\lambda x. e$
| $e_1 e_2$

Similar to simplifying $(x+1) * (2x-2)$
using middle-school algebra

What are the rewrite rules for λ -calculus?

Operational semantics

$$(\lambda x . t_1) t_2 \rightarrow [x \mapsto t_2]t_1$$

β -reduction
(function call)

$[x \mapsto t_2]t_1$ means “ t_1 with all **free occurrences** of x replaced with t_2 ”

```
incl(int x) {  
    return x+1  
}
```

$$(\lambda x . x + 1) 2 \rightarrow [x \mapsto 2]x + 1 = 3$$

```
incl(2);
```

$$[x \mapsto y]\lambda x . x = \lambda x . y \quad \times$$

What does free occurrences mean?

Semantics: β -reduction

$$(\lambda x . t_1) t_2 \rightarrow [x \mapsto t_2]t_1$$

β -reduction
(function call)

$[x \mapsto t_2]t_1$ means “ t_1 with all **free occurrences** of x replaced with t_2 ”

The core of β -reduction reduces to substitution:

$$[x \mapsto s]x = s$$

$$[x \mapsto s]y = y (x \neq y)$$

$$[x \mapsto s]\lambda y . t_1 = \lambda y . [x \mapsto s]t_1 (y \neq x \wedge y \notin FV(s))$$

$$[x \mapsto s]t_1 t_2 = [x \mapsto s]t_1 [x \mapsto s]t_2$$

Semantics: α -renaming

$$\lambda x . e =_{\alpha} \lambda y . [x \mapsto y]e$$

- Rename a formal parameter and replace all its occurrences in the body

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

$$[x \mapsto y]\lambda x . x = \lambda x . y \quad \times$$

$$[x \mapsto y]\lambda x . x =_{\alpha} [x \mapsto y]\lambda z . z = \lambda z . z \quad \checkmark$$

Call-by-name v.s. Call-by-value

$$(\lambda x . e_1) e_2 =_{\text{name}} [x \mapsto e_2]e_1$$

Call-by-Name: From leftmost/outermost, allowing **no reductions** inside abstractions.

$$(\lambda x . e_1) e_2 =_{\text{value}} [x \mapsto [e_2]]e_1$$

Call-by-Value: only when its right-hand side has already been reduced to a value—a term that **cannot be reduced any further**

Currying: multiple arguments

$$\lambda(x, y) . e = \lambda x . \lambda y . e$$

$$(\lambda(x, y) . x + y) \ 2 \ 3 =$$

$$(\lambda x . \lambda y . x + y) \ 2 \ 3 = (\lambda y . 2 + y) \ 3 = [y \mapsto 3]2 + y = 5$$

Transformation of multi-arguments functions to higher-order functions is called currying (in the honor of Haskell Curry)



What about the others?

- ~~Assignment~~
- ~~Booleans, integers, characters, strings, ...~~
- ~~Conditionals~~
- ~~Loops~~
- ~~Functions~~
- ~~Recursion~~
- ~~References / pointers~~
- ~~Objects and classes~~
- ~~Inheritance~~

λ -calculus:Booleans

- How do we encode Boolean values (**TRUE** and **FALSE**) as functions?
- What do we do with Boolean?
- Make a binary choice
 - if b then e1 else e2

Booleans: API

We need to define three functions

- `let TRUE = ???`
- `let FALSE = ???`
- `let ITE = $\lambda b\ x\ y \rightarrow ???$ -- if b then x else y`

such that

- `ITE TRUE apple banana = apple`
- `ITE FALSE apple banana = banana`

Booleans: implementation

Boolean implementation

- `let TRUE = $\lambda x y. x$` -- Returns its first argument
- `let FALSE = $\lambda x y. y$` -- Returns its second argument
- `let ITE = $\lambda b x y. b x y$` -- Applies condition to branches

Why they are correct?

Booleans: examples

eval ite_true:

```
ITE TRUE e1 e2
= (λb x y. b x y) TRUE e1 e2  -- expand def ITE
=β (λx y. TRUE x y) e1 e2  -- beta-step
=β (λy. TRUE e1 y) e2  -- beta-step
=β TRUE e1 e2  -- expand def TRUE
= (λx y. x) e1 e2  -- beta-step
=β (λy. e1) e2  -- beta-step
=β e1
```

Other boolean API:

```
let NOT = λb. ITE b FALSE TRUE
```

```
let AND = λb1 b2. ITE b1 b2 FALSE
```

```
let OR = λb1 b2. ITE b1 TRUE b2
```

TODOs by next lecture

- Install OCaml on your laptop
- Start to work on HW1
- Come to the discussion session if you have questions