CS 162 Programming languages

Lecture 3: λ-calculus II

Yu Feng

Winter 2021

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

$$
\begin{aligned}
e ::= \ & x \\
| \ & \lambda x.\ e \\
| \ & e_1\ e_2
\end{aligned}
$$

$\backslash x \rightarrow e$ (Haskell)

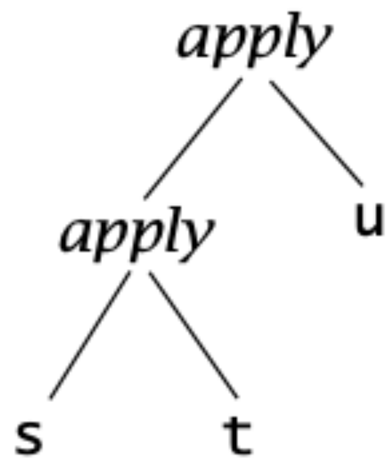
fun $x \rightarrow e$ (OCaml)

lambda $x.\ e$ (λ+)

- 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

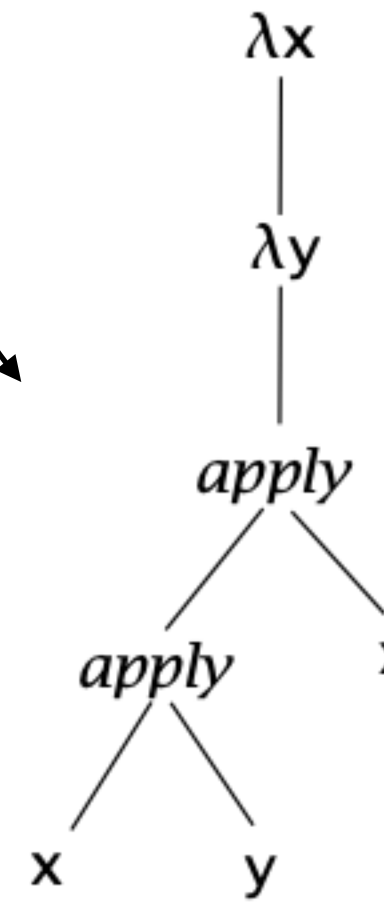
Precedence

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



Application associates to the **left**

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



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 λx. e

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

λx. x

λx. (λy. x)

x is bounded

x y

λy. x y

(λx. λy. 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:

$FV(x) = x$	$FV(x\ y) = \{x, y\}$
$FV(\lambda x.\ e) = FV(e) \setminus x$	$FV(\lambda y.\ x\ y) = \{x\}$
$FV(e_1\ e_2) = FV(e_1) \cup FV(e_2)$	$FV((\lambda x.\ \lambda y.\ y)\ x) = \{x\}$

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

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

$$
\begin{aligned}
e ::= \; & x \\
\mid \; & \lambda x.\, e \\
\mid \; & e_1 \; e_2
\end{aligned}
$$

What are the rewrite rules for λ-calculus?

Operational semantics

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

β-reduction
(function call)

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

```
inc1(int x ) {
  return x+1
}

inc1(2);
```

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

$$[x \mapsto y]\lambda x\,.\,x = \lambda x\,.\,y$$

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

$$\lambda x\,.\,x =_\alpha \lambda y\,.\,y =_\alpha \lambda z\,.\,z$$

$$[x \mapsto y]\lambda x\,.\,x = \lambda x\,.\,y$$

$$[x \mapsto y]\lambda x\,.\,x =_\alpha [x \mapsto y]\lambda z\,.\,z = \lambda z\,.\,z$$

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

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

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

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

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 = λb x y -> ??? -- if b then x else y

such that

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

Booleans: implementation

Boolean implementation

- let TRUE = λx y. x -- Returns its first argument
- let FALSE = λx y. y -- Returns its second argument
- let ITE = λb x y. b x y -- Applies condition to branches

Why they are correct?

Booleans: examples

eval ite_true:

$$\begin{aligned}
& \text{ITE TRUE } e_1\ e_2 \\
&= (\lambda b\ x\ y.\ b\ \ x\ \ y)\ \ \text{TRUE}\ \ e_1\ e_2 && \text{-- expand def ITE} \\
&=_\beta\ (\lambda x\ y.\ \text{TRUE}\ x\ \ y)\ \ e_1\ e_2 && \text{-- beta-step} \\
&=_\beta\ (\lambda y.\ \text{TRUE}\ e_1\ y)\ \ e_2 && \text{-- beta-step} \\
&=_\beta\ \text{TRUE}\ e_1\ e_2 && \text{-- expand def TRUE} \\
&= (\lambda x\ y.\ \ x)\ e_1\ e_2 && \text{-- beta-step} \\
&=_\beta\ (\lambda y.\ e_1)\ \ e_2 && \text{-- beta-step} \\
&=_\beta\ e_1
\end{aligned}$$

Other boolean API:

let NOT = λb. ITE b FALSE TRUE

let AND = $\lambda b_1\ b_2.$ ITE $b_1\ b_2$ FALSE

let OR = $\lambda b_1\ b_2.$ ITE b_1 TRUE b_2

TODOs by next lecture

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