Lambda Calculus : Syntax

CS3100 Fall 2019

Review

Last time

- Higher Order Functions

Today

- Lambda Calculus: Basis of FP!
  - Origin, Syntax, substitution, alpha equivalence

Computability

In 1930s

- What does it mean for the function $f : \mathbb{N} \rightarrow \mathbb{N}$ to be *computable*?
- **Informal definition:** A function is computable if using pencil-and-paper you can compute $f(n)$ for any n.
- Three different researchers attempted to formalise *computability*.

Alan Turning

- Defined an idealised computer -- **The Turing Machine** (1935)
- A function is computable if and only if it can be computed by a turning machine
- A programming language is turing complete if:
  - It can map every turing machine to a program.
  - A program can be written to emulate a turing machine.
  - It is a superset of a known turning complete language.

Alonzo Church

- Developed the **λ-calculus** as a formal system for mathematical logic (1929 - 1932).
- Postulated that a function is computable (in the intuitive sense) if and only if it can be written as a lambda term (1935).

- Church was Turing's PhD advisor!
- Turing showed that the systems defined by Church and his system were equivalent.
  - **Church-Turing Thesis**

Kurt Gödel

- Defined the class of **general recursive functions** as the smallest set of functions containing
  - all the constant functions
  - the successor function and
  - closed under certain operations (such as compositions and recursion).
- He postulated that a function is computable (in the intuitive sense) if and only if it is general recursive.

Impact of Church-Turing thesis

- The **“Church-Turing Thesis”** is by itself is one of the most important ideas on computer science
  - The impact of Church and Turing’s models goes far beyond the thesis itself.

Impact of Church-Turing thesis

- Oddly, however, the impact of each has been in almost completely separate communities
  - Turing Machines ⇒ Algorithms & Complexity

  - Lambda Calculus ⇒ Programming Languages
- Not accidental
  - Turing machines are quite low level ⇒ well suited for measuring resources (**efficiency**).
  - Lambda Calculus is quite high level ⇒ well suited for abstraction and composition (**structure**).

Programming Language Expressiveness

- So what language features are needed to express all computable functions?
  - *What's the minimal language that is Turing Complete?*
- Observe that many features that we have seen in this class were syntactic sugar
  - **Multi-argument functions** - simulate using partial application
  - **For loop, while loop** - simulate using recursive functions
  - **Mutable heaps** - simulate using functional maps and pass around.

Functional Heap

In [1]:

```
type ('k,'v) heap = 'k -> 'v option

let empty_heap : ('k,'v) heap = fun k -> None

let set (h : ('k,'v) heap) (x : 'k) (v : 'v) : ('k,'v) heap =
  fun k -> if k = x then Some v else h k

let get (h : ('k,'v) heap) (x : 'k) : 'v option = h x
```

```
Findlib has been successfully loaded. Additional directive
s:
  #require "package";;      to load a package
  #list;;                   to list the available packages
  #camlp4o;;                to load camlp4 (standard synta
x)
  #camlp4r;;                to load camlp4 (revised syntax)
  #predicates "p,q,...";;   to set these predicates
  Topfind.reset();;         to force that packages will be
reloaded
  #thread;;                 to enable threads
```

Out[1]:

```
type ('k, 'v) heap = 'k -> 'v option
```

Out[1]:

```
val empty_heap : ('k, 'v) heap = <fun>
```

Out[1]:

```
val set : ('k, 'v) heap -> 'k -> 'v -> ('k, 'v) heap = <fun
>
```

Out[1]:

```
val get : ('k, 'v) heap -> 'k -> 'v option = <fun>
```

Functional Heap

In [2]:

```
let _ =
  let h = set empty_heap "a" 0 in
  let h = set h "b" 1 in
  (get h "a", get h "b", get h "c")
```

Out[2]:

```
- : int option * int option * int option = (Some 0, Some 1,
None)
```

- You can imagine passing around the heap as an **implicit extra argument** to every function.
  - The issue of storing values of different types, default values, etc. can be orthogonally addressed.

All you need is ~~Love~~ *Functions.*

Lambda Calculus : Syntax

$$\begin{array}{rcll} e & ::= & x & \text{(Variable)} \\ & | & \lambda x.\, e & \text{(Abstraction)} \\ & | & e\ e & \text{(Application)} \end{array}$$

- This grammar describes ASTs; not for parsing (ambiguous!)
- Lambda expressions also known as lambda **terms**
- $\lambda x.\, e$ is like `fun x -> e`

That's it! Nothing but higher order functions

Why Study Lambda Calculus?

- It is a "core" language
  - Very small but still Turing complete
- But with it can explore general ideas
  - Language features, semantics, proof systems, algorithms, ...
- Plus, higher-order, anonymous functions (aka lambdas) are now very popular!

- C++ (C++11), PHP (PHP 5.3.0), C# (C# v2.0), Delphi (since 2009), Objective C, Java 8, Swift, Python, Ruby (Procs), ...
- and functional languages like OCaml, Haskell, F#, ...

Three Conventions

1. Scope of λ extends as far right as possible
   - Subject to scope delimited by parentheses
   - $\lambda x. \lambda y. x\ y$ is the same as $\lambda x. (\lambda y. (x\ y))$

2. Function Application is left-associative
   - `x y z` is `(x y) z`
   - Same rule as OCaml

3. As a convenience, we use the following syntactic sugar for local declarations
   - `let x = e1 in e2` is short for $(\lambda x. e2)\ e1$.

Lambda calculus interpreter in OCaml

- In Assignment 2, you will be implementing a lambda calculus interpreter in OCaml.
- What is the Abstract Syntax Tree (AST)?

```
type expr =
   | Var of string
   | Lam of string * expr
   | App of expr * expr
```

Lambda expressions in OCaml

- y is `Var "y"`
- $\lambda x. x$ is `Lam ("x", Var "x")`
- $\lambda x. \lambda y. x\ y$ is `Lam ("x",(Lam("y",App (Var "x", Var "y"))))`
- $(\lambda x. \lambda y. x\ y)\ \lambda x. x\ x$ is

```
App
  (Lam ("x", Lam ("y",App (Var "x", Var "y"))),
   Lam ("x", App (Var "x", Var "x")))
```

In [3]:

```
#use "init.ml";;
```

```
val parse : string -> Syntax.expr = <fun>
val free_variables : string -> Eval.SS.elt list = <fun>
val substitute : string -> string -> string -> string = <fu
n>
```

In [4]:

```
parse "y";;
parse "\\x.x";;
parse "\\x.\\y.x y";;
parse "(\\x.\\y.x y) \\x. x x";;
```

Out[4]:

```
- : Syntax.expr = Var "y"
```

Out[4]:

```
- : Syntax.expr = Lam ("x", Var "x")
```

Out[4]:

```
- : Syntax.expr = Lam ("x", Lam ("y", App (Var "x", Var
"y")))
```

Out[4]:

```
- : Syntax.expr =
App (Lam ("x", Lam ("y", App (Var "x", Var "y"))),
 Lam ("x", App (Var "x", Var "x")))
```

Quiz 1

$\lambda x.(y\ z)$ and $\lambda x.\ y\ z$ are equivalent.

1. True
2. False

Quiz 1

$\lambda x.(y\ z)$ and $\lambda x.\ y\ z$ are equivalent.

1. True ✅

2. False

Quiz 2

What is this term's AST? $\lambda x.\, x\ x$

1. `App (Lam ("x", Var "x"), Var "x")`
2. `Lam (Var "x", Var "x", Var "x")`
3. `Lam ("x", App (Var "x", Var "x"))`
4. `App (Lam ("x", App ("x", "x")))`

Quiz 2

What is this term's AST? $\lambda x.\, x\ x$

1. `App (Lam ("x", Var "x"), Var "x")`
2. `Lam (Var "x", Var "x", Var "x")`
3. `Lam ("x", App (Var "x", Var "x"))` ✅
4. `App (Lam ("x", App ("x", "x")))`

Quiz 3

This term is equivalent to which of the following?

$\lambda x.\, x\ a\ b$

1. $(\lambda x.\, x)\ (a\ b)$
2. $(((\lambda x.\, x)\ a)\ b)$
3. $\lambda x.\, (x\ (a\ b))$
4. $(\lambda x.\, ((x\ a)\ b))$

Quiz 3

This term is equivalent to which of the following?

$\lambda x.\, x\ a\ b$

1. $(\lambda x.\, x)\ (a\ b)$
2. $(((\lambda x.\, x)\ a)\ b)$

3. $\lambda x. (x\ (a\ b))$
4. $(\lambda x. ((x\ a)\ b))$ ✅

Free Variables

In

```
λx. x y
```

- The first `x` is the binder.
- The second `x` is a **bound** variable.
- The `y` is a **free** variable.

Free Variables

Let $FV(t)$ denote the free variables in a term t.

We can define $FV(t)$ inductively over the definition of terms as follows:

$$\begin{array}{rcl} FV(x) & = & \{x\} \\ FV(\lambda x. t_1) & = & FV(t_1) \setminus \{x\} \\ FV(t_1\ t_2) & = & FV(t_1) \cup FV(t_2) \end{array}$$

If $FV(t) = \varnothing$ then we say that t is a **closed** term.

Quiz 4

What are the free variables in the following?

1. $\lambda x. x\ (\lambda y. y)$
2. $x\ y\ z$
3. $\lambda x. (\lambda y. y)\ x\ y$
4. $\lambda x. (\lambda y. x)\ y$

Quiz 4

What are the free variables in the following?

1. $\lambda x. x\ (\lambda y. y)$ $\{\}$
2. $x\ y\ z$ $\{x, y, z\}$
3. $\lambda x. (\lambda y. y)\ x\ y$ $\{y\}$
4. $\lambda x. (\lambda y. x)\ y$ $\{y\}$

In [5]:

```
free_variables "\\x.x (\\y. y)";;
free_variables "x y z";;
free_variables "\\x.(\\y. y) x y";;
free_variables "\\x.(\\y.x) y";;
```

Out[5]:

```
- : Eval.SS.elt list = []
```

Out[5]:

```
- : Eval.SS.elt list = ["x"; "y"; "z"]
```

Out[5]:

```
- : Eval.SS.elt list = ["y"]
```

Out[5]:

```
- : Eval.SS.elt list = ["y"]
```

α-equivalence

Lambda calculus uses **static scoping** (just like OCaml)

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

This is equivalent to:

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

- Renaming bound variables consistently preserves meaning
  - This is called as **α-renaming** or **α-conversion.**
- If a term t_1 is obtained by α-renaming another term t_2 then t_1 and t_2 are said to be **α-equivalent.**

Quiz 5

Which of the following equivalences hold?

1. $\lambda x.\, x\ (\lambda y.\, y)\ y =_\alpha \lambda y.\, y\ (\lambda x.\, x)\ x$
2. $\lambda x.\, x\ (\lambda y.\, y)\ y =_\alpha \lambda y.\, y\ (\lambda x.\, x)\ y$
3. $(\lambda x.\, x\ (\lambda y.\, y)\ y) =_\alpha \lambda w.\, w\ (\lambda w.\, w)\ y$

Quiz 5

Which of the following equivalences hold?

1. $\lambda x.\, x\ (\lambda y.\, y)\ y =_\alpha \lambda y.\, y\ (\lambda x.\, x)\ x$ ❌
2. $\lambda x.\, x\ (\lambda y.\, y)\ y =_\alpha \lambda y.\, y\ (\lambda x.\, x)\ y$ ❌
3. $\lambda x.\, x\ (\lambda y.\, y)\ y =_\alpha \lambda w.\, w\ (\lambda w.\, w)\ y$ ✅

Substitution

- In order to formally define α-equivalence, we need to define **substitutions**.
- Substitution replaces **free** occurrences of a variable x with a lambda term N in some other term M.
  - We write it as $M[N/x]$. (read "N for x in M").

For example,

$$(\lambda x.\, x\ y)[(\lambda z.\, z)/y] = \lambda x.\, x\ (\lambda z.\, z)$$

Substitution is quite subtle. So we will start with our intuitions and see how things break and finally work up to the correct example.

Substitution: Take 1

$$\begin{array}{rcll} x[s/x] & = & s & \\ y[s/x] & = & y & \text{if } x \neq y \\ (\lambda y.\, t_1)[s/x] & = & \lambda y.\, t_1[s/x] & \\ (t_1\ t_2)[s/x] & = & (t_1[s/x])\ (t_2[s/x]) & \end{array}$$

This definition works for most examples. For example,

$$(\lambda y.\, x)[(\lambda z.\, z\ w)/x] = \lambda y.\, \lambda z.\, z\ w$$

Substitution: Take 1

$$\begin{array}{rcll} x[s/x] & = & s & \\ y[s/x] & = & y & \text{if } x \neq y \\ (\lambda y.\, t_1)[s/x] & = & \lambda y.\, t_1[s/x] & \\ (t_1\ t_2)[s/x] & = & (t_1[s/x])\ (t_2[s/x]) & \end{array}$$

However, it fails if the substitution is on the bound variable:

$$(\lambda x.\, x)[y/x] = \lambda x.\, y$$

The **identity** function has become a **constant** function!

Substitution: Take 2

$$\begin{array}{rcll} x[s/x] & = & s & \\ y[s/x] & = & y & \text{if } x \neq y \\ (\lambda x.\, t_1)[s/x] & = & \lambda x.\, t_1 & \\ (\lambda y.\, t_1)[s/x] & = & \lambda y.\, t_1[s/x] & \text{if } x \neq y \\ (t_1\ t_2)[s/x] & = & (t_1[s/x])\ (t_2[s/x]) & \end{array}$$

However, this is not quite right. For example,

$$(\lambda x.\, y)[x/y] = \lambda x.\, x$$

- The **constant** function has become a **identity** function.
- The problem here is that the free x gets **captured** by the binder x.

Substitution: Take 3

Capture-avoiding substitution

$$\begin{array}{rcll} x[s/x] & = & s & \\ y[s/x] & = & y & \text{if } x \neq y \\ (\lambda x.\, t_1)[s/x] & = & \lambda x.\, t_1 & \\ (\lambda y.\, t_1)[s/x] & = & \lambda y.\, t_1[s/x] & \text{if } x \neq y \text{ and } y \notin FV(s) \\ (t_1\ t_2)[s/x] & = & (t_1[s/x])\ (t_2[s/x]) & \end{array}$$

- Unfortunately, this made substitution a partial function
  - There is no valid rule for $(\lambda x.\, y)[x/y]$

Substitution: Take 4

Capture-avoiding substitution + totality

$$\begin{array}{rcll} x[s/x] & = & s & \\ y[s/x] & = & y & \text{if } x \neq y \\ (\lambda x.\, t_1)[s/x] & = & \lambda x.\, t_1 & \\ (\lambda y.\, t_1)[s/x] & = & \lambda y.\, t_1[s/x] & \text{if } x \neq y \text{ and } y \notin FV(s) \\ (\lambda y.\, t_1)[s/x] & = & \lambda w.\, t_1[w/y][s/x] & \text{if } x \neq y \text{ and } y \in FV(s) \text{ and } w \text{ is fresh} \\ (t_1\ t_2)[s/x] & = & (t_1[s/x])\ (t_2[s/x]) & \end{array}$$

- A **fresh** binder is different from every other binder in use **(generativity)**.
- In the case above,

$$w \text{ is fresh } \equiv w \notin FV(t_1) \cup FV(s) \cup \{x\}$$

Now our example works out:

$$(\lambda x.\, y)[x/y] = \lambda w.\, x$$

In [6]:

```
substitute "\\y.x" "x" "\\z.z w"
```

Out[6]:

```
- : string = "λy.λz.z w"
```

In [7]:

```
substitute "\\x.x" "x" "y"
```

Out[7]:

```
- : string = "λx.x"
```

In [8]:

```
substitute "\\x.y" "y" "x"
```

Out[8]:

```
- : string = "λx0.x"
```

α-equivalence formally

$=_\alpha$ is an equivalence (reflexive, transitive, symmetric) relation such that:

$$\frac{}{x =_\alpha x} \qquad \frac{M =_\alpha M' \quad N =_\alpha N'}{M\ N =_\alpha M'\ N'}$$

$$\frac{z \notin FV(M) \cup FV(N) \quad M[z/x] =_\alpha N[z/y]}{\lambda x.\, M =_\alpha \lambda y.\, N}$$

Convention

From now on,

- Unless stated otherwise, we identify lambda terms up to α-equivalence.
  - when we speak of lambda terms being **equal**, we mean that they are α-equivalent

Fin.