

CSE 130: Spring 2011 Programming Languages

The λ -Calculus



Background

- Developed in 1930's by [Alonzo Church](#)
 - studied in [logic](#) and computer science
- Test bed for programming languages
 - Simple, Powerful, Extensible

"Whatever the next 700 languages turn out to be, they will surely be variants of lambda calculus" (Landin '66)

Syntax

- Three kinds of expressions (terms):
 - $e ::= x$ Variables
 - | $\lambda x. e$ Functions (λ -abstraction)
 - | $e_1 e_2$ Application
- $\lambda x. e$ is a one-argument function with body e
- $e_1 e_2$ is a function application
- Application associates to the left:
 - $x y z$ means $(x y) z$
- Abstraction extends to the right as far as possible:
 - $\lambda x. x \lambda y. x y z$ means $\lambda x. x (\lambda y. ((x y) z))$

Examples of Lambda Expressions

- The [identity](#) function:
 - $I =_{\text{def}} \lambda x. x$
- A function that given an argument y discards it and returns the identity function:
 - $\lambda y. (\lambda x. x)$ | $\lambda y. (I)$
 $\lambda x. x$
- A function that given a function f invokes it on the identity function:
 - $\lambda f. f (\lambda x. x)$ | $\lambda f. (f I)$

Free and Bound Variables

- Variable “shadowing”
 - Different occurrences of var may refer to different values
- E.g., in ML: $\text{let } x = E \text{ in } x + (\text{let } x = E' \text{ in } x) + x$

- In lambda calculus: $\lambda x. x (\lambda x. x) x$


Renaming Bound Variables

α -renaming:

- λ -terms after renaming bound var occurrences
- considered identical to original

Example: $\lambda x. x$ is identical to $\lambda y. y$ and to $\lambda z. z$

Rename bound variables so names unique

- e.g., write $\lambda x. x (\lambda y. y) x$ instead of $\lambda x. x (\lambda x. x) x$
- Easy to see the scope of bindings

Substitution

$[E'/x] E$: Substitution of E' for x in E

- Uniquely rename bound vars in E and E'
- Do textual substitution of E' for x in E

Example: $[y (\lambda x. x)/x] \lambda y. (\lambda x. x) y x$

- After renaming: $[y (\lambda v. v)/x] \lambda z. (\lambda u. u) z x$
- After substitution: $\lambda z. (\lambda u. u) z (y (\lambda v. v))$

Informal Semantics

The evaluation of $(\lambda x. e) e'$ $[e'/x]e$

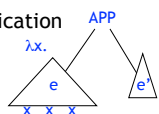
- binds x to e'
- evaluates e with the new binding
- yields the result of this evaluation

Like $\text{let } x = e' \text{ in } e$

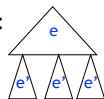
Example: $(\lambda f. f (f e)) g$
evaluates to $g (g e)$

Another View of Reduction

The application



becomes:



Terms can *grow*
substantially by *reduction*

Examples of Evaluation

- The identity function:

$(\lambda x. x) E$
 $\Rightarrow [E / x] x$
 $\Rightarrow E$

- Another example with the identity:

$(\lambda f. f (\lambda x. x)) (\lambda x. x)$
 $\Rightarrow [\lambda x. x / f] f (\lambda x. x)$
 $\Rightarrow [(\lambda x. x) / f] f (\lambda x. x)$
 $\Rightarrow \lambda x. x (\lambda x. x)$
 $\Rightarrow [\lambda x. x / x] x$
 $\Rightarrow \lambda x. x$

Examples of Evaluation

$(\lambda x. x x)(\lambda y. y y)$
 $\Rightarrow [\lambda y. y y / x] x x$
 $\Rightarrow (\lambda y. y y)(\lambda y. y y)$
 $\Rightarrow (\lambda x. x x)(\lambda y. y y)$
 $\Rightarrow \dots$

A non-terminating evaluation !

Next

“Programming” with the λ -Calculus

Programming with the λ -calculus

How does the λ -calculus relate to “real” programming languages ?

- Booleans / If-then-else ?
- Records
- Integers ?
- Recursion ?
- Functions (well, those we have ...)

Encoding Booleans in λ -calculus

Q: What can we do with a boolean?

A: Make a **binary choice**

Q: So, how can you view this as a “function” ?

A: Bool = fun that takes **two** choices, returns **one**

- $\text{true} =_{\text{def}} \lambda x. \lambda y. x$
- $\text{false} =_{\text{def}} \lambda x. \lambda y. y$
- if E_1 then E_2 else $E_3 =_{\text{def}} E_1 E_2 E_3$

Example: “if **true** then **u** else **v**” is

$(\lambda x. \lambda y. x) u v @_{\beta} (\lambda y. u) v @_{\beta} u$

Boolean Operations

Boolean operations: **not**

Function takes **b**:

returns function takes **x,y**:

returns “opposite” of **b**’s return

$\text{not} =_{\text{def}} \lambda b. (\lambda x. \lambda y. b y x)$

Boolean operations: **or**

Function takes **b₁, b₂**:

returns function takes **x,y**:

returns (if **b₁** then **x** else (if **b₂** then **x** else **y**))

$\text{or} =_{\text{def}} \lambda b_1. \lambda b_2. (\lambda x. \lambda y. b_1 x (b_2 x y))$

Programming with the λ -calculus

How does the λ -calculus relate to “real” programming languages ?

- Booleans / If-then-else ?
- Records
- Integers ?
- Recursion ?
- Functions (well, those we have ...)

Encoding Pairs (and so, Records)

Q: What can we do with a pair ?

A: We can select one of its elements

Pair: function takes a bool,
returns the left or the right element

$mkpair \equiv \lambda x. \lambda y. \lambda b. b x y$

Note: "pair" encoded as λ -abstraction, "waiting" for bool

$fst\ p \equiv_{def} p\ true$ $fst \equiv \lambda p. p\ true$

$snd\ p \equiv_{def} p\ false$ $snd \equiv \lambda p. p\ false$

Ex: $fst\ (mkpair\ x\ y) \ @\ (mkpair\ x\ y)\ true \ @\ true\ x\ y \ @\ x$

Programming with the λ -calculus

How does the λ -calculus relate to
"real" programming languages ?

- Booleans / If-then-else ?
- Records
- Integers ?
- Recursion ?
- Functions (well, those we have ...)

Encoding Natural Numbers

Q: What can we do with a natural number ?

A: Iterate a number of times over some function

Nat: function that takes fun f , starting value s :
returns: f applied to s a number of times

$0 \equiv_{def} \lambda f. \lambda s. s$ $0 \equiv_{def} \lambda f. \lambda s. s$

$1 \equiv_{def} \lambda f. \lambda s. f\ s$ $1 \equiv_{def} \lambda f. \lambda s. f\ s$

$2 \equiv_{def} \lambda f. \lambda s. f\ (f\ s)$ $2 \equiv_{def} \lambda f. \lambda s. f\ (f\ s)$

$M \equiv_{def} \lambda f. \lambda s. f\ (f\ (f\ (f\ s)))$

Called Church numerals, unary representation

Note: $(n\ f\ s)$: apply f to s "n" times, i.e. $f^n(s)$

Operating on Natural Numbers

- Testing equality with 0

$iszero\ n \equiv_{def} n\ (\lambda b. false)\ true$

$iszero \equiv_{def} \lambda n. (\lambda b. false)\ true$

- The successor function

$succ\ n \equiv_{def} \lambda f. \lambda s. f\ (n\ f\ s)$

$succ \equiv_{def} \lambda n. \lambda f. \lambda s. f\ (n\ f\ s)$

- Addition

$add \equiv_{def} \lambda n_1. \lambda n_2. n_1\ succ\ n_2$

- Multiplication

$mult \equiv_{def} \lambda n_1. \lambda n_2. n_1\ (add\ n_2)\ 0$

Ex: Computing with Naturals

What is the result of `add 0` ?

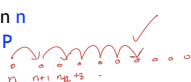
$$\begin{aligned}
 & (\lambda n_1. \lambda n_2. n_1 \text{ succ } n_2) 0 \\
 \Rightarrow_{\beta} & \lambda n_2. 0 \text{ succ } n_2 = \\
 \Rightarrow_{\beta} & \lambda n_2. (\lambda f. \lambda s. s) \text{ succ } n_2 \\
 \Rightarrow_{\beta} & \lambda n_2. n_2 = \\
 \Rightarrow_{\beta} & \lambda x. x
 \end{aligned}$$

Programming with the λ -calculus

How does the λ -calculus relate to “real” programming languages ?

- Booleans / If-then-else ?
- Records
- Integers ?
- Recursion ?
- Functions (well, those we have ...)

Encoding Recursion

- Write a function `find` that:
takes “predicate” `P`, “natural” `n` returns:
 smallest natural
 larger than `n`
 satisfying `P`
- find P n* 
- `find` can encode all recursion...
 - ...but how to write it ?

Encoding Recursion

`find` satisfies the equation:

$$\text{find } p \ n = \text{if } p \ n \text{ then } n \text{ else } \text{find } p \ (\text{succ } n)$$

- Define: $F = \lambda f. \lambda p. \lambda n. (p \ n) \ n \ (f \ p \ (\text{succ } n))$
 - A fixpoint of F is an x s.t. $x = F \ x$
 - `find` is a fixpoint of F !
 - as `find p n = F find p n`
 - so `find = F find`
- $f_{\text{oo}} \equiv F \ f_{\text{oo}}$
 $\equiv \lambda p. \lambda n. \text{if } (p \ n) \text{ then } n \text{ else } (f_{\text{oo}} \ p \ (\text{succ } n))$*

The Y-Combinator

Define: $Y \stackrel{\text{def}}{=} \lambda F. (\lambda y. F(y y)) (\lambda x. F(x x))$

- Called the **fixpoint combinator** as...
 - $Y F$

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

$$== F ((\lambda x. F (x x)) (\lambda z. F (z z)))$$

$$== F (Y F)$$
 - i.e. $Y F =_{\beta} F (Y F)$
- Can get fixpoint **for any** λ -calculus function

Whoa!

Define: $F = \lambda f. \lambda p. \lambda n. (p\ n)\ n\ (f\ p\ (\text{succ}\ n))$
and: $\text{find} = Y\ F$

Whats going on ?

$$\begin{aligned} & \text{find } p \ n \\ &=_{\beta} Y \ F \ p \ n \\ &=_{\beta} F \ (\underline{Y \ F}) \ p \ n \\ &=_{\beta} F \ \text{find } p \ n \\ &=_{\beta} (p \ n) \ n \ (\text{find } p \ (\text{succ } n)) \end{aligned}$$

Many other fixpoint combinators

Including those that work for CBV

Including Klop's Combinator:

$$Y_k \stackrel{\text{def}}{=} (\underbrace{L L L L L L L L L L L L L L L L L L L L L L}_{k \text{ times}})$$

where:

```
L =def λaλbλcλdλeλfλgλhλiλjλkλlλmλnλoλpλqλsλtλuλvλwλxλyλzλr.  
r(thisisafixpointcombinator)
```

Programming with the λ -calculus

How does the λ -calculus relate to “real” programming languages ?

- Booleans / If-then-else ?
- Records
- Integers ?
- Recursion ?
- Functions (well, those we have ...)

Thats all folks!

- Hope 130 taught you something ...
 - ... many ways of computational thinking
- Good luck for final
 - On Monday
 - Review Session: Sun 5-7, CSE 4140
- Want more?
 - CSE 230 (Winter 2012)