CSE 230

The λ-Calculus

Background

Developed in 1930's by Alonzo Church

Studied in logic and computer science

Test bed for procedural and functional PLs

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**

$\mid\ \lambda x.e$ **Functions (λ-abstraction)**

$\mid\ e_1\ e_2$ **Application**

Syntax

Application associates to the left

$x\ y\ z$ means $(x\ y)\ z$

Abstraction extends as far right as possible:

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

Examples of Lambda Expressions

Identity function

$$I =_{def} \lambda x.\ x$$

A function that always returns the identity fun

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

A function that applies arg to identity function:

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

Scope of an Identifier (Variable)

“part of program where variable is accessible”

Free and Bound Variables

$\lambda x.\ E$ **Abstraction binds variable** x **in** E

- x is the newly introduced variable
- E is the scope of x
- x is bound in $\lambda x.\ E$

Free and Bound Variables

y is free in E if it occurs *not bound* in E

$$\text{Free}(x) = \{x\}$$

$$\text{Free}(E_1\ E_2) = \text{Free}(E_1) \cup \text{Free}(E_2)$$

$$\text{Free}(\lambda x.\ E) = \text{Free}(E) - \{x\}$$

e.g: $\text{Free}(\lambda x.\ x\ (\lambda y.\ x\ y\ z)) = \{ z \}$

Renaming Bound Variables

α-renaming

λ-terms after renaming bound variables

Considered identical to original

Example: $\lambda x.\ x == \lambda y.\ y == \lambda z.\ z$

Rename bound variables so names unique

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

1. Uniquely rename bound vars in E and E'
2. Do textual substitution of E' for x in E

Example: [y (λx. x)/x] λy. (λx. x) y x

1. After renaming: [y (λv. v)/x] λz. (λu. u) z x
2. After substitution: λz. (λu. u) z (y (λv. v))

Semantics ("Evaluation")

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

1. binds x to e'
2. evaluates e with the new binding
3. yields the result of this evaluation

Semantics: Beta-Reduction

$$(\lambda x.\ e)\ e' \rightarrow [e'/x]e$$

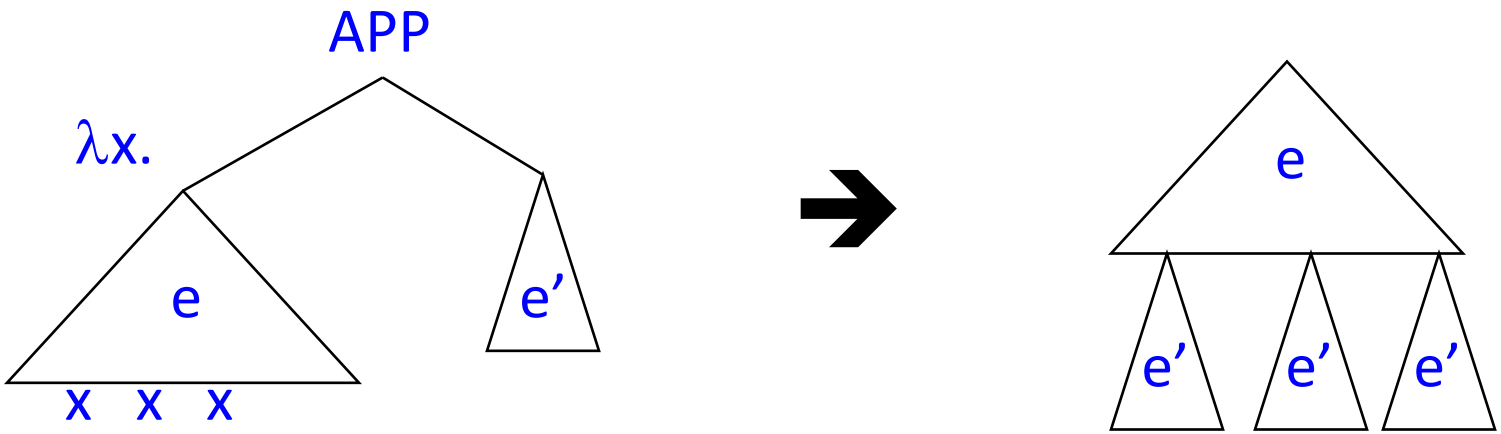
Semantics ("Evaluation")

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

1. binds x to e'
2. evaluates e with the new binding
3. yields the result of this evaluation

Example: $(\lambda f.\ f\ (f\ e))\ g \rightarrow g\ (g\ e)$

Another View of Reduction



Terms can grow substantially by reduction

Examples of Evaluation

Identity function

$$
\begin{aligned}
& (\lambda x.\ x)\ E \\
\rightarrow\ & [E\ /\ x]\ x \\
=\ & E
\end{aligned}
$$

Examples of Evaluation

… yet again

$$
\begin{aligned}
& (\lambda f.\ f\ (\lambda x.\ x))\ (\lambda x.\ x) \\
\rightarrow\ & [\lambda x.\ x\ /\ f]\ f\ (\lambda x.\ x) \\
=\ & [(\lambda x.\ x)\ /\ f]\ f\ (\lambda y.\ y) \\
=\ & (\lambda x.\ x)\ (\lambda y.\ y) \\
\rightarrow\ & [\lambda y.\ y\ /x]\ x \\
=\ & \lambda y.\ y
\end{aligned}
$$

Examples of Evaluation

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

$$\rightarrow [\lambda y.\ y\ y\ /\ x]\ x\ x$$

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

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

$$\rightarrow \ldots$$

A non-terminating evaluation !

Review

A calculus of functions:

$$e := x \mid \lambda x.\, e \mid e_1\, e_2$$

Eval strategies = "Where to reduce" ?

Normal, Call-by-name, Call-by-value

Church-Rosser Theorem

Regardless of strategy, upto one "normal form"

Programming with the λ-calculus

λ-calculus vs. "real languages" ?

Local variables?

Bools , If-then-else ?

Records?

Integers ?

Recursion ?

Functions: well, those we have …

Local Variables (Let Bindings)

$$\textbf{let } x = e_1 \textbf{ in } e_2$$

is just

$$(\lambda x.\ e_2)\ e_1$$

Programming with the λ-calculus

λ-calculus vs. "real languages" ?

Local variables (YES!)

Bools , If-then-else ?

Records?

Integers ?

Recursion ?

Functions: well, those we have …

Encoding Booleans in λ-calculus

What can we do with a boolean?

Make *a binary choice*

How can you view this as a "function" ?

Bool is a *fun* that takes *two* choices, returns *one*

Encoding Booleans in λ-calculus

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

$$\text{true} =_{\text{def}} \lambda x.\ \lambda y.\ x$$

$$\text{false} =_{\text{def}} \lambda x.\ \lambda y.\ y$$

$$\text{if } E_1 \text{ then } E_2 \text{ else } E_3 =_{\text{def}} E_1\ E_2\ E_3$$

Example: "if true then u else v" is

$$(\lambda x.\ \lambda y.\ x)\ u\ v \rightarrow (\lambda y.\ u)\ v \rightarrow u$$

Boolean Operations: Not, Or

Boolean operations: not

Function takes b:

returns function takes x,y:

returns "opposite" of b's return

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

Boolean operations: or

Function takes b_1, b_2:

returns function takes x,y:

returns (if b_1 then x else (if b_2 then x else y))

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

Programming with the λ-calculus

λ-calculus vs. "real languages" ?

Local variables (YES!)

Bools , If-then-else (YES!)

Records?

Integers ?

Recursion ?

Functions: well, those we have …

Encoding Pairs (and so, Records)

What can we do with a pair ?

Select one of its elements

Pair = function takes a bool,

returns the left or the right element

$\text{mkpair}\ e_1\ e_2 =_{def} \lambda b.\ b\ e_1\ e$

= "function-waiting-for-bool"

$\text{fst}\ p =_{def} p\ \text{true}$

$\text{snd}\ p =_{def} p\ \text{false}$

Encoding Pairs (and so, Records)

mkpair e_1 e_2 $=_{def}$ λb. b e_1 e

fst p $=_{def}$ p true

snd p $=_{def}$ p false

Example

fst (mkpair x y) $\rightarrow$ (mkpair x y) true $\rightarrow$ true x y $\rightarrow$ x

Programming with the λ-calculus

λ-calculus vs. "real languages" ?

Local variables (YES!)

Bools , If-then-else (YES!)

Records (YES!)

Integers ?

Recursion ?

Functions: well, those we have …

Encoding Natural Numbers

What can we do with a natural number ?

Iterate a *number* of times over some function

n = function that takes fun f, starting value s,

returns: f applied to s "n" times

$$0 =_{def} \lambda f.\ \lambda s.\ s$$
$$1 =_{def} \lambda f.\ \lambda s.\ f\ s$$
$$2 =_{def} \lambda f.\ \lambda s.\ f\ (f\ s)$$
$$\vdots$$

Called Church numerals (Unary Representation)

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

Operating on Natural Numbers

Testing equality with 0

$\text{iszero } n =_{def} n\ (\lambda b.\ \text{false})\ \text{true}$

$\text{iszero} =_{def} \lambda n.(\lambda\ b.\text{false})\ \text{true}$

Successor function

$\text{succ } n =_{def} \lambda f.\ \lambda s.\ f\ (n\ f\ s)$

$\text{succ} =_{def} \lambda n.\ \lambda f.\ \lambda s.\ f\ (n\ f\ s)$

Addition

$\text{add } n_1\ n_2 =_{def} n_1\ \text{succ}\ n_2$

$\text{add} =_{def} \lambda n_1.\lambda n_2.\ n_1\ \text{succ}\ n_2$

Multiplication

$\text{mult } n_1\ n_2 =_{def} n_1\ (\text{add } n_2)\ 0$

$\text{mult} =_{def} \lambda n_1.\lambda n_2.\ n_1\ (\text{add } n_2)\ 0$

Example: Computing with Naturals

What is the result of add 0 ?

$(\lambda n_1.\ \lambda n_2.\ n_1\ \text{succ}\ n_2)\ 0\ \rightarrow$

$\lambda n_2.\ 0\ \text{succ}\ n_2 =$

$\lambda n_2.\ (\lambda f.\ \lambda s.\ s)\ \text{succ}\ n_2 \rightarrow$

$\lambda n_2.\ n_2 =$

$\lambda x.\ x$

Example: Computing with Naturals

mult 2 2

$\rightarrow$ 2 (add 2) 0

$\rightarrow$(add 2) ((add 2) 0)

$\rightarrow$2 succ (add 2 0)

$\rightarrow$ 2 succ (2 succ 0)

$\rightarrow$ succ (succ (succ (succ 0)))

$\rightarrow$ succ (succ (succ (λf. λs. f (0 f s))))

$\rightarrow$ succ (succ (succ (λf. λs. f s)))

$\rightarrow$ succ (succ (λg. λy. g ((λf. λs. f s) g y)))

$\rightarrow$ succ (succ (λg. λy. g (g y)))

$\rightarrow$* λg. λy. g (g (g (g y)))

= 4

Programming with the λ-calculus

λ-calculus vs. “real languages” ?

Local variables (YES!)

Bools , If-then-else (YES!)

Records (YES!)

Integers (YES!)

Recursion ?

Functions: well, those we have …

Encoding Recursion

Write a function find:

IN : predicate **P**, number **n**

OUT: *smallest num* >= **n** s.t. **P(n)=True**

Encoding Recursion

find satisfies the equation:

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

- Define: F = λf.λp.λn.$\boxed{\text{(p n)}}\boxed{\text{n}}\boxed{\text{(f p (succ n))}}$
- A fixpoint of F is an x s.t. x = F x
- find is a fixpoint of F !
  - as <u>find</u> p n = <u>F find</u> p n
  - so find = F find

Q: Given λ-term F, how to write its fixpoint ?

The Y-Combinator

Fixpoint Combinator

$Y =_{def} \lambda F.\ (\lambda y.F(y\ y))\ (\lambda x.\ F(x\ x))$

Earns its name as …

$Y\ F\ \rightarrow\ (\lambda y.F\ (y\ y))\ (\lambda x.\ F\ (x\ x))$

$\quad\quad \rightarrow\ F\ ((\lambda x.F\ (x\ x))(\lambda z.\ F\ (z\ z)))\ \leftarrow\ F\ (Y\ F)$

So, for any λ-calculus function F get Y F is fixpoint!

$Y\ F = F\ (Y\ F)$

Whoa!

Define: $F = \lambda f.\lambda p.\lambda n.(p\ n)\ n\ (f\ p\ (succ\ n))$

and: $find = Y\ F$

Whats going on ?

$find\ p\ n$

$=_\beta Y\ F\ p\ n$

$=_\beta F\ (Y\ F)\ p\ n$

$=_\beta F\ find\ p\ n$

$=_\beta (p\ n)\ n\ (find\ p\ (succ\ n))$

Y-Combinator in Practice

```
fac n = if n<1 then 1 else n * fac (n-1)
```

is just

```
fac = \n->if n<1 then 1 else n * fac (n-1)
```

is just

```
fac = Y (\f n->if n<1 then 1 else n*f(n-1))
```

All Recursion Factored Into Y

Many other fixpoint combinators

Including Klop's Combinator:

$\mathbf{Y_k} =_{\text{def}}$ (L L L L L L L L L L L L L L L L L L L L L L L L L L)

where:

$\mathbf{L} =_{\text{def}}$ λabcdefghIjklmnopqstuvwxyzr.

r (t h i s i s a f i x p o i n t c o m b i n a t o r)

Expressiveness of λ-calculus

Encodings are fun

Programming in pure λ-calculus is not!

We know λ-calculus encodes them

So add 0,1,2,…,true,false,if-then-else to PL

Next, **types**…