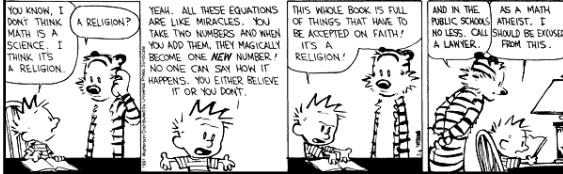


# Lambda Calculus

Prof. Evan Chang  
Meeting 13, CSCI 3155, Fall 2009



## Announcements

- Project 1 (two-week assignment)
  - Due Thu Oct 15
  - Checkpoint Due Thu Oct 8

2

## Finish Structural Induction

$\forall l \in \text{'a list}. P(l)$   
**Structural Induction**

datatype 'a list = nil | :: of 'a \* 'a list

*Base case* (circled)  
*Inductive case* (circled)

- ① Base Cases (non-recursive cases of our datatype)  
Show  $P(\text{nil})$  holds
- ② Inductive Cases  
 $l = h :: t$  Assume  $P(t)$  holds  
Show  $P(h :: t)$  holds

4

## How about with datatypes?

```
datatype 'a list = nil | :: of 'a * 'a list
fun append (nil, k) = k
  | append (h :: t, k) = h :: append (t, k)
```

Theorem: For values  $l : \text{'a list}$  and  $k : \text{'a list}$ ,  
 $\text{append } (l, k) \Downarrow v$  (for some value  $v$ )

Proof: By induction on the structure of  $l$

Base Case:  $l = \text{nil}$   
 $\text{append } (\text{nil}, k) \rightarrow k$  ( $k \Downarrow v$  bc  $k$ )

Inductive Case:  $l = h :: t$   
 $\text{append } (h :: t, k) \rightarrow h :: \text{append } (t, k)$   
 $\rightarrow h :: v$  *By i.h.,  $\Downarrow v$  (for some  $v$ )*

## Example: append is associative

```
fun @ (nil, k) = k
  | @ (h :: t, k) = h :: (t @ k)
```

Theorem: For values  $l_1, l_2, l_3$  of type  $\text{ty list}$ ,  $(l_1 @ l_2) @ l_3 \equiv l_1 @ (l_2 @ l_3)$

$e \Downarrow v$  iff  $e' \Downarrow v$   
and  $e' \Downarrow v$ , then  $e \Downarrow v$

$[e \Downarrow v \text{ iff } e' \Downarrow v]$

6

*datatype nat = Zero | Succ of nat*

**Example: append is associative**

```
fun @ (nil, k) = k
  | @ (h :: t, k) = h :: (t @ k)
```

Theorem: For values  $l_1, l_2, l_3$  of type `ty list`,  $(l_1 @ l_2) @ l_3 \equiv l_1 @ (l_2 @ l_3)$

*Proof: By structural induction on  $l_1$*

*Base Case  $l_1 = \text{nil}$*

$(\text{nil} @ l_2) @ l_3 \rightarrow l_2 @ l_3 \rightarrow l_{23}$

$\text{nil} @ (l_2 @ l_3) \rightarrow l_2 @ l_3 \rightarrow l_{23}$

*Inductive Case  $l_1 = h :: t_1$*

$((h :: t_1) @ l_2) @ l_3 \rightarrow (h :: (t_1 @ l_2)) @ l_3$

$\rightarrow h :: ((t_1 @ l_2) @ l_3)$

$\rightarrow h :: (t_1 @ (l_2 @ l_3))$

$\rightarrow (h :: t_1) @ (l_2 @ l_3)$

*By i.h.  $(t_1 @ l_2) @ l_3 \equiv t_1 @ (l_2 @ l_3)$*

7

*$[h:] = (h :: \text{nil})$*

**Exercise: rev' computes rev**

```
fun rev nil = nil | rev (h :: t) = rev t @ [h]
fun rev' (nil, acc) = acc
  | rev' (h :: t, acc) = rev' (t, h :: acc)
```

Theorem: For values  $l, k$ : `ty list`,  $(\text{rev } l) @ k \equiv \text{rev}' (l, k)$

*Proof: By structural induction on the structure of  $l$*

*Base Case:  $l = \text{nil}$*

$(\text{rev nil}) @ k \rightarrow \text{nil} @ k \rightarrow k$

$(\text{rev}' \text{ nil}, k) \rightarrow k$

*Inductive Case:  $l = h :: t_1$*

$(\text{rev } (h :: t_1)) @ k \rightarrow (\text{rev } t_1 @ [h]) @ k$

$\rightarrow \text{rev } t_1 @ (h :: k)$

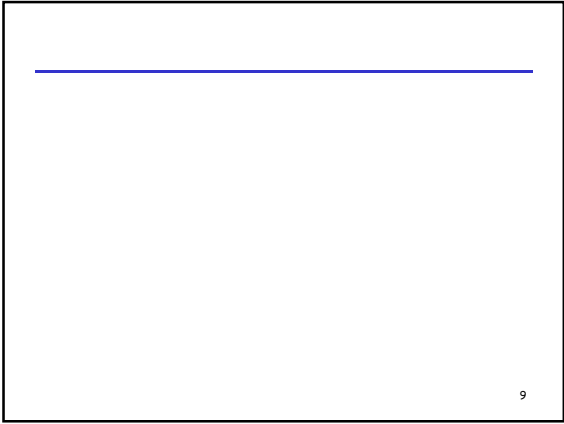
$\rightarrow \text{rev } t_1 @ (h :: k)$

$(\text{rev}' (h :: t_1), k) \rightarrow \text{rev}' (t_1, h :: k)$

*By assoc. of @*

*By ind. hyp  $(\text{rev } t_1) @ (h :: k) \equiv \text{rev}' (t_1, h :: k)$*

8



## Lambda Calculus

### Lambda Calculus

---

**Goal:** Come up with a "core" language that's as small as possible and still Turing complete

This will give a way of illustrating important language features and algorithms

11

### Lambda Background

---

- Developed in 1930's by **Alonzo Church**
- Subsequently studied by many people
  - Still studied today!
- Considered the "testbed" for procedural and functional languages
  - Simple
  - Powerful
  - Easy to extend with new features of interest
  - "Lambda:PL :: Turing Machine:Complexity"

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

12

## Lambda Syntax

- The  $\lambda$ -calculus has 3 kinds of expressions (terms)

$e ::= x$  Variables  
|  $\lambda x. e$  Functions (abstractions)  
|  $e_1 e_2$  Application

- $\lambda x. e$  is a one-argument anonymous function with body  $e = (f \ x \Rightarrow e)$
- $e_1 e_2$  is a function application
- Application associates to the left  
 $x y z ::= (x y) z$
- Abstraction extends as far to the right as possible  
 $\lambda x. x \lambda y. x y z ::= \lambda x. (x [\lambda y. \{(x y) z\}])$

13

## Examples of Lambda Expressions

- The identity function:

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

- A function that, given an argument  $y$ , discards it and yields the identity function:

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

- A function that, given a function  $f$ , invokes it on the identity function:

$\lambda f. f I$

14

## Examples of Lambda Expressions

- The identity function:

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

- A function that, given an argument  $y$ , discards it and yields the identity function:

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

- A function that, given a function  $f$ , invokes it on the identity function:

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

15

## Scope of Variables

- As in all languages with variables, it is important to discuss the notion of scope
  - The scope of an identifier is the portion of a program where the identifier is accessible
- An abstraction  $\lambda x. E$  binds variable  $x$  in  $E$ 
  - $x$  is the newly introduced variable
  - $E$  is the scope of  $x$  (unless  $x$  is shadowed)
  - We say  $x$  is bound in  $\lambda x. E$
  - Just like formal function arguments are bound in the function body

16

## Free Your Mind!

- Just as in any language with statically-nested scoping we have to worry about variable shadowing
  - An occurrence of a variable might refer to different things in different contexts
- Example let-expressions (as in ML):  
 $\text{let } x = 5 \text{ in } x + (\text{let } x = 9 \text{ in } x) + x$
- In  $\lambda$ -calculus:

$\lambda x. x (\lambda \underline{x}. \underline{x}) x$

17

## Renaming Bound Variables

=  $\alpha$ -renaming or  $\alpha$ -conversion

- $\lambda$ -terms that can be obtained from one another by renaming bound variables are considered identical
- This is called  $\alpha$ -equivalence
- Ex:  $\lambda x. x$  is identical to  $\lambda y. y$  and to  $\lambda z. z$
- Intuition:
  - By changing the name of a formal argument and all of its occurrences in the function body, the behavior of the function *does not change*
  - In  $\lambda$ -calculus such functions are considered identical

18

## Make It Easy On Yourself !

- Convention: we will always try to rename bound variables so that they are all unique
  - e.g., write  $\lambda x. x (\lambda y. y) x$  instead of  $\lambda x. x (\lambda x. x) x$
- This makes it easy to see the scope of bindings and also prevents confusion!

19

## Substitution

- The substitution of  $e'$  for  $x$  in  $e$  (written  $[e'/x]e$ )
  - Step 1. Rename bound variables in  $e$  and  $e'$  so they are unique
  - Step 2. Perform the textual substitution of  $e'$  for  $x$  in  $e$
- Called capture-avoiding substitution

20

## Substitution

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

21

## Substitution

- Example:  $[y (\lambda x. x) / x] \lambda y. (\lambda x. x) y x$ 
  - After renaming:  $[y (\lambda x. x) / x] \lambda z. (\lambda u. u) z x$
  - After substitution:  $\lambda z. (\lambda u. u) z (y (\lambda x. x))$
- If we are not careful with scopes we might get:  
 $\lambda y. (\lambda x. x) y (y (\lambda x. x)) \leftarrow \text{wrong!}$

22

## Informal Semantics

- All we've got are functions, so all we can do is call them!

23

## Informal Semantics

- All we've got are functions, so all we can do is call them!
- The evaluation of  $(\lambda x. e) e'$ 
  - Binds  $x$  to  $e'$
  - Evaluates  $e$  with the new binding
  - Yields the result of this evaluation
- Like a function call, or like "let  $x = e'$  in  $e$ "
- Example:  
 $(\lambda f. f (f e)) g$  evaluates to  $g (g e)$

24

## Informal Semantics

- All we've got are functions, so all we can do is call them!
- The evaluation of  $(\lambda x. e) e'$ 
  - Binds  $x$  to  $e'$
  - Evaluates  $e$  with the new binding
  - Yields the result of this evaluation
- Like a function call, or like "let  $x = e'$  in  $e$ "
- Example:  
 $(\lambda f. f (f e)) g$  evaluates to  $g (g e)$

25

## Operational Semantics

### beta-reduction

$$(\lambda x. e_1) e_2 \rightarrow_{\beta} [e_2/x]e_1$$

capture avoiding substitution

26

## How is $\lambda$ -calculus related to "real life"?

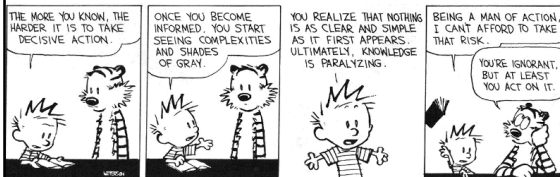
## Functional Programming

- The  $\lambda$ -calculus is a prototypical functional language with:
  - no side effects
  - several evaluation strategies
  - lots of functions
  - nothing but functions (pure  $\lambda$ -calculus does not have any other data type)
- How can we program with functions?
- How can we program with **only** functions?

28

## How Complex Is Lambda?

- Given  $e_1$  and  $e_2$ , how complex (a la CS theory) is it to determine if:  
 $e_1 \rightarrow_{\beta}^* e$  and  $e_2 \rightarrow_{\beta}^* e$



## Expressiveness of $\lambda$ -Calculus

- The  $\lambda$ -calculus is a minimal system but can express
  - data types (integers, booleans, lists, trees, etc.)
  - branching, recursion
- This is enough to **encode Turing machines**
  - We say the lambda calculus is **Turing-complete**
  - Corollary:  $e_1 =_{\beta} e_2$  is **undecidable**
- That means we can encode any computation we want in it ... if we're sufficiently clever ...

30

## Encodings

- Still, how do we encode all these constructs using only functions?
- Idea: encode the "behavior" of values and not their structure

31

## Encoding Booleans in $\lambda$ -Calculus

- What can we **do** with a boolean?
  - we can **make a binary choice** (= "if" exp)
- A boolean is a function that, given two choices, selects one of them:
  - **true**  $=_{\text{def}} \lambda x. \lambda y. x$
  - **false**  $=_{\text{def}} \lambda x. \lambda y. y$
  - **if  $E_1$  then  $E_2$  else  $E_3$**   $=_{\text{def}}$

32

## Encoding Booleans in $\lambda$ -Calculus

- What can we **do** with a boolean?
  - we can **make a binary choice** (= "if" exp)
- A boolean is a function that, given two choices, selects one of them:
  - **true**  $=_{\text{def}} \lambda x. \lambda y. x$
  - **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  $(\lambda x. \lambda y. x) u v \rightarrow_{\beta} (\lambda y. u) v \rightarrow_{\beta} u$

33

## Let's try to define **or**

- Recall:
  - **true**  $=_{\text{def}} \lambda x. \lambda y. x$
  - **false**  $=_{\text{def}} \lambda x. \lambda y. y$
  - **if  $E_1$  then  $E_2$  else  $E_3$**   $=_{\text{def}} E_1 E_2 E_3$
- Intuition:
  - **or a b = if a then true else b**
- Either of these will work:
  - **or**  $=_{\text{def}} \lambda a. \lambda b. a \text{ true } b$
  - **or**  $=_{\text{def}} \lambda a. \lambda b. \lambda x. \lambda y. a x (b x y)$

34

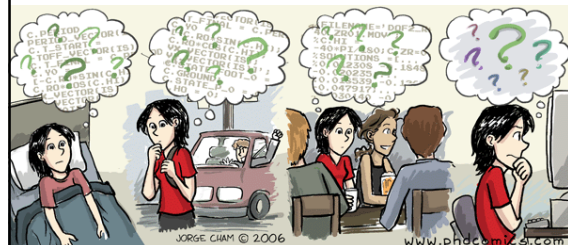
## This is getting painful to check ...

- Let's use SML ...

35

## More Boolean Encodings

- Think about how to do **and** and **not**
- Without peeking!



## Encoding and and not

- **and**  $a\ b$  = if  $a$  then  $b$  else  $false$ 
  - **and** =<sub>def</sub>  $\lambda a. \lambda b. a\ b\ false$
  - **and** =<sub>def</sub>  $\lambda a. \lambda b. \lambda x. \lambda y. a\ (b\ x\ y)$
- **not**  $a$  = if  $a$  then  $false$  else  $true$ 
  - **not** =<sub>def</sub>  $\lambda a. a\ false\ true$
  - **not** =<sub>def</sub>  $\lambda a. \lambda x. \lambda y. a\ y\ x$

37

## Encoding Pairs in $\lambda$ -Calculus

- What can we **do** with a pair?
  - we can **access one of its elements** (= "field access")
- A pair is a function that, given a boolean, returns the first or second element
  - mkpair**  $x\ y$  =<sub>def</sub>  $\lambda b. b\ x\ y$
  - fst**  $p$  =<sub>def</sub>  $p\ true$
  - snd**  $p$  =<sub>def</sub>  $p\ false$
- $fst\ (mkpair\ x\ y) \rightarrow_{\beta} (mkpair\ x\ y)\ true$   
 $\rightarrow_{\beta} true\ x\ y \rightarrow_{\beta} x$

38

## Encoding Numbers $\lambda$ -Calculus

- What can we **do** with a natural number?
  - we can **iterate a number of times** over some function (= "for loop")
- A natural number is a function that given an operation  $f$  and a starting value  $z$ , applies  $f$  a number of times to  $z$ :
  - $0$  =<sub>def</sub>  $\lambda f. \lambda z. z$
  - $1$  =<sub>def</sub>  $\lambda f. \lambda z. f\ z$
  - $2$  =<sub>def</sub>  $\lambda f. \lambda z. f\ (f\ z)$
  - Very similar to `List.fold_left` and friends
- These are numerals in a unary representation
- Called **Church numerals**

39

## Computing with Natural Numbers

- The successor function
  - succ**  $n$  =<sub>def</sub>  $\lambda f. \lambda s. f\ (n\ f\ s)$
  - or **succ**  $n$  =<sub>def</sub>  $\lambda f. \lambda s. n\ f\ (f\ s)$
- Addition
  - plus**  $n_1\ n_2$  =<sub>def</sub>  $n_1\ succ\ n_2$
- Multiplication
  - mult**  $n_1\ n_2$  =<sub>def</sub>  $n_1\ (add\ n_2)\ 0$
- Testing equality with 0
  - iszero**  $n$  =<sub>def</sub>  $n\ (\lambda b. false)\ true$
- Subtraction
  - Is not instructive, but makes a fun exercise ...

40

## Computation Example

- What is the result of the application **add 0**?
  - $(\lambda n_1. \lambda n_2. n_1\ succ\ n_2)\ 0 \rightarrow_{\beta}$
  - $\lambda n_2. 0\ succ\ n_2 =$
  - $\lambda n_2. (\lambda f. \lambda s. s)\ succ\ n_2 \rightarrow_{\beta}$
  - $\lambda n_2. n_2 =$
  - $\lambda x. x$
- By computing with functions we can express some optimizations
  - But we need to **reduce under the lambda**
  - Thus this "never" happens in practice

41

## Toward Recursion

- Given a predicate  $p$ , encode the function "**find**" such that "**find**  $p\ n$ " is the smallest natural number which is at least  $n$  and satisfies  $p$
- Ideas? How do we begin?

42

## Encoding Recursion

- Given a predicate  $p$ , encode the function "find" such that "find  $p$   $n$ " is the smallest natural number which is at least  $n$  and satisfies  $p$

- find satisfies the equation  
$$\text{find } p \ n = \text{if } p \ n \ \text{then } n \ \text{else } \text{find } p \ (\text{succ } n)$$

- Define

$$F = \lambda f. \lambda p. \lambda n. (p \ n) \ n \ (f \ p \ (\text{succ } n))$$

- We need a fixed point of  $F$

$$\text{find} = F \ \text{find}$$

or

$$\text{find } p \ n = F \ \text{find } p \ n$$

43

## The Fixed-Point Combinator $Y$

- Let  $Y = \lambda F. (\lambda y. F(y \ y)) (\lambda x. F(x \ x))$

- This is called the **fixed-point combinator**

- Verify that  $Y \ F$  is a fixed point of  $F$

$$Y \ F \rightarrow_{\beta} (\lambda y. F(y \ y)) (\lambda x. F(x \ x)) \rightarrow_{\beta} F \ (Y \ F)$$

- Thus  $Y \ F =_{\beta} F \ (Y \ F)$

- Given any function in  $\lambda$ -calculus we can compute its fixed-point (!)

- Thus we can define "find" as the fixed-point of the function  $F$  from the previous slide

- Essence of recursion is the self-application " $y \ y$ "

44

## For Next Time

- Reading
- Online discussion forum
  - $\geq 1$  substantive question, comment, or answer each week
- Work on project

45