### 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

---

### Structural Induction

**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 list, \( (l_1 @ l_2) @ l_3 \equiv l_1 @ (l_2 @ l_3) \)

Proof:

\[ e = e' \text{ if } e \equiv v, \text{ then } e' \equiv v \text{ and } e' \equiv v \]

\[ e \equiv v \text{ if } e \equiv v' \]

---

**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 \( k : \text{'a list} \),  
\[ \text{append}(l, k) \Downarrow v \text{ (for some value v)} \]

Proof:

- **Base case**  
  \[ \text{append}(\text{nil}, k) \Downarrow k \]

- **Inductive case**  
  \[ \text{append}(\text{h :: t}, k) \Downarrow \text{h :: append}(\text{t}, k) \]

---

**Finish Structural Induction**
Example: append is associative

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

Theorem: For values l₁, l₂, l₃ of type ty list, (l₁ @ l₂) @ l₃ ≅ l₁ @ (l₂ @ l₃)

Exercise: rev' computes rev

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

Theorem: For values l, k : t list, (rev l) @ k ≅ rev' (l, k)

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

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)
Lambda Syntax
- The λ-calculus has 3 kinds of expressions (terms):
  - Variables $e ::= x$
  - Functions (abstractions) $\lambda x. e$
  - Application $e_1 e_2$

Examples of Lambda Expressions
- The identity function:
  - $I = \lambda x. x$
- 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:
  - $\lambda f. f I$

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.

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):
  - let $x = 5$ in $x + (\text{let } x = 2 \text{ in } x) + x$
- In λ-calculus:
  - $\lambda x. x (\lambda x. x) x$

Renaming Bound Variables
- λ-terms that can be obtained from one another by renaming bound variables are considered identical.
- This is called α-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 λ-calculus such functions are considered identical.
Make It Easy On Yourself

• Convention: we will always try to rename bound variables so that they are all unique
  - e.g., write $\lambda x. (\lambda y. y) x$ instead of $\lambda x. (\lambda x. x) x$

• This makes it easy to see the scope of bindings and also prevents confusion!

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

Substitution

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

• If we are not careful with scopes we might get:
  $\lambda y. (\lambda x. x) y (y (\lambda x. x)) \leftarrow$ wrong!

Informal Semantics

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

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 $g (g e)$
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 \(g (g e)\)

Operational Semantics

- \(\text{beta-reduction}\)
  \[(\lambda x. e_1) e_2 \rightarrow^\beta [e_2/x]e_1\]
  - Capture avoiding substitution

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?

How is \(\lambda\)-calculus related to "real life"?

- 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: \(e_1 \equiv^* e_2\) is undecidable
- That means we can encode any computation we want in it ... if we're sufficiently clever ...
Encodings

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

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 \(E_3\) \(= \text{def} E_1 E_2 E_3\)

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 \(a\ b = \text{if } a \text{ then true else } b\)
  - Either of these will work:
    - or \(= \text{def} \lambda a. \lambda b. a \text{ true } b\)
    - or \(= \text{def} \lambda a. \lambda b. \lambda x. \lambda y. a \ x \ (b \ y)\)

This is getting painful to check ...

• Let's use SML ...

More Boolean Encodings

• Think about how to do and and not
• Without peeking!
Encoding and and not

- \( \text{and} \ a \ b = \text{if } a \text{ then } b \text{ else false} \)
- \( \text{and} \equiv \lambda a. \lambda b. \ a \ b \ false \)
- \( \text{and} = \lambda \lambda \lambda \ a. \lambda \lambda \ lambda \ x. \ lambda \ y. \ a \ (b \times y) \ y \)

- \( \text{not} a = \text{if } a \text{ then false else true} \)
- \( \text{not} \equiv \lambda a. \ a \ false \ true \)
- \( \text{not} = \lambda \lambda \lambda \ a. \lambda \ x. \lambda \ y. \ a \ y \ x \)

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
  \( \text{mkpair } x \ y = \lambda a. \ a \ b \times y \)
- fst \( p \) = \( \lambda \) \( p \ true \)
- snd \( p \) = \( \lambda \) \( p \ false \)
- \( \text{fst} (\text{mkpair } x \ y) \rightarrow (\text{mkpair } x \ y) \ true \rightarrow x \)
- \( \text{snd} (\text{mkpair } x \ y) \rightarrow (\text{mkpair } x \ y) \ false \rightarrow y \)

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 \( z \):
  - \( 0 = \lambda f. \lambda z. z \)
  - \( 1 = \lambda f. \lambda z. f z \)
  - \( 2 = \lambda f. \lambda z. f (f z) \)
  - Very similar to List fold_left and friends
- These are numerals in a unary representation
- Called Church numerals

Computing with Natural Numbers

- The successor function
  - \( \text{succ} \ n \equiv \lambda f. \lambda s. \ f (n \ f \ s) \)
  - \( \text{succ} n \rightarrow \lambda f. \lambda s. \ n \ f (f \ s) \)
- Addition
  - \( \text{plus} \ n_1 \ n_2 \equiv \lambda f. \ n_1 \ fn_2 \)
- Multiplication
  - \( \text{mult} \ n_1 \ n_2 \equiv \lambda f. \ n_1 \ (\text{add} \ n_2) \ f \)
- Testing equality with 0
  - \( \text{iszero} \ n \equiv \lambda b. \ n \ false \ true \)
- Subtraction
  - Is not instructive, but makes a fun exercise ...

Computation Example

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

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?
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
  \[
  \text{find } p \ n = \text{if } p \ n \text{ then } n \text{ else find } (\text{succ } n)
  \]
- 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
  \[
  \text{find } p \ n = F \ \text{find } p \ n
  \]

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 \( F \)
  \[
  Y \ F \rightarrow (\lambda y. F (y \ y)) (\lambda x. F (x \ x)) \rightarrow F (Y \ F)
  \]
  - Thus \( Y \ F = Y \ F (Y \ F) \)
- 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 "\( y \ y \)"

For Next Time

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