Motivation

- Commonly-used programming languages are large and complex
  - ANSI C99 standard: 538 pages
  - ANSI C++ standard: 714 pages
  - Java language specification 2.0: 505 pages
- Not good vehicles for understanding language features or explaining program analysis

Goal

- Develop a “core language” that has
  - The essential features
  - No overlapping constructs
  - And none of the cruft
    - Extra features of full language can be defined in terms of the core language (“syntactic sugar”)
- Lambda calculus
  - Standard core language for single-threaded procedural programming
  - Often with added features (e.g., state); we’ll see that later

Lambda Calculus is Practical!

- An 8-bit microcontroller (Zilog Z8 encore board w/4KB SRAM) computing 1 + 1 using Church numerals in the Lambda calculus

Tim Fraser
Origins of Lambda Calculus

• Invented in 1936 by Alonzo Church (1903-1995)
  - Princeton Mathematician
  - Lectures of lambda calculus published in 1941
  - Also know for
    - Church’s Thesis
    - All effective computation is expressed by recursive (decidable) functions, i.e., in the lambda calculus
    - Church's Theorem
    - First order logic is undecidable

Syntax:

- $e ::= x$ variable
- $\lambda x.e$ function abstraction
- $e e$ function application

Only constructs in pure lambda calculus

- Functions take functions as arguments and return functions as results
- I.e., the lambda calculus supports higher-order functions

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

- Bind $x$ to $e_2$
- Evaluate $e_1$
- Return the result of the evaluation

This is called “beta-reduction”

- $(\lambda x.e_1) e_2 \rightarrow_{\beta} e_1[e_2/x]$
- $(\lambda x.e_1) e_2$ is called a redex
- We’ll usually omit the beta

Three Conveniences

- Syntactic sugar for local declarations
  - let $x = e_1$ in $e_2$ is short for $(\lambda x.e_2) e_1$

- Scope of $\lambda$ extends as far to the right as possible
  - $\lambda x.\lambda y.x y$ is $\lambda x.(\lambda y.(x y))$

- Function application is left-associative
  - $x y z$ is $(x y) z$
Scoping and Parameter Passing

• Beta-reduction is not yet precise
  - \((\lambda x. e_1) \ e_2 \rightarrow e_1[e_2/x]\)
  - what if there are multiple \(x\)'s?

• Example:
  - let \(x = a\) in
  - let \(y = \lambda z. x\) in
  - let \(x = b\) in \(y\)
  - which \(x\)'s are bound to \(a\), and which to \(b\)?

Free Variables and Alpha Conversion

• The set of free variables of a term is
  
  \[ FV(x) = \{x\} \]
  
  \[ FV(\lambda x. e) = FV(e) - \{x\} \]
  
  \[ FV(e_1 \ e_2) = FV(e_1) \cup FV(e_2) \]

• A term \(e\) is closed if \(FV(e) = \emptyset\)

• A variable that is not free is bound

Static (Lexical) Scope

• Just like most languages, a variable refers to the closest definition

• Make this precise using variable renaming
  - The term
    - let \(x = a\) in let \(y = \lambda z. x\) in let \(x = b\) in \(y\)
  - is “the same” as
    - let \(x = a\) in let \(y = \lambda z. x\) in let \(w = b\) in \(y w\)
  - Variable names don’t matter

Alpha Conversion

• Terms are equivalent up to renaming of bound variables
  - \(\lambda x. e = \lambda y. (e[y/x])\) if \(y \notin FV(e)\)

• This is often called \(alpha\) conversion, and we will use it implicitly whenever we need to avoid capturing variables when we perform substitution
Substitution

- Formal definition:
  - $x[e/x] = e$
  - $z[e/x] = z$ if $z \neq x$
  - $(e_1 e_2)[e/x] = (e_1[e/x] e_2[e/x])$
  - $\lambda z.e_1[e/x] = \lambda z.(e_1[e/x])$ if $z \neq x$ and $z \notin \text{FV}(e)$

- Example:
  - $(\lambda x.y x) x = \alpha (\lambda w.y w) x \rightarrow \beta y x$
  - (We won’t write alpha conversion down in the future)

A Note on Substitutions

- People write substitution many different ways
  - $e_1[e_2/x]$
  - $e_1[x\rightarrow e_2]$
  - $[x/e_2]e_1$
  - and more...

- But they all mean the same thing

Multi-Argument Functions

- We can’t (yet) write multi-argument functions
  - E.g., a function of two arguments $\lambda (x, y).e$
  - Trick: Take arguments one at a time
    - $\lambda x.\lambda y.e$
    - This is a function that, given argument $x$, returns a function that, given argument $y$, returns $e$
    - $(\lambda x.\lambda y.e) a b \rightarrow (\lambda y.e[a/x]) b \rightarrow e[a/x][b/y]$
  - This is often called Currying and can be used to represent functions with any # of arguments

Booleans

- true = $\lambda x.\lambda y.x$
- false = $\lambda x.\lambda y.y$
- if a then b else c = a b c

- Example:
  - if true then b else c = $(\lambda x.\lambda y.x) b c \rightarrow (\lambda y.b) c \rightarrow b$
  - if false then b else c = $(\lambda x.\lambda y.y) b c \rightarrow (\lambda y.y) c \rightarrow c$
Combinators

• Any closed term is also called a combinator
  ▪ So true and false are both combinators

• Other popular combinators
  ▪ I = λx.x
  ▪ S = λx.λy.x
  ▪ K = λx.λy.λz.x z (y z)
  ▪ Can also define calculi in terms of combinators
    - E.g., the SKI calculus
    - Turns out the SKI calculus is also Turing complete

Pairs

• (a, b) = λx.if x then a else b
• fst = λp.p true
• snd = λp.p false

• Then
  ▪ fst (a, b) →* a
  ▪ snd (a, b) →* b

Natural Numbers (Church)

• 0 = λx.λy.y
• 1 = λx.λy.x y
• 2 = λx.λy.x(x y)
• i.e., n = λx.λy.<apply x n times to y>
• succ = λz.λx.λy.x(z x y)
• iszero = λz.z (λy.false) true

Natural Numbers (Scott)

• 0 = λx.λy.x
• 1 = λx.λy.y 0
• 2 = λx.λy.y 1
• i.e., n = λx.λy.y (n-1)
• succ = λz.λx.λy.y z
• pred = λz.z 0 (λx.x)
• iszero = λz.z true (λx.false)
A Nonderministic Semantics

$(\lambda x. e_1) e_2 \rightarrow e_1[e_2/x]$

$e \rightarrow e'$

$(\lambda x. e) \rightarrow (\lambda x. e')$

$e_1 \rightarrow e_1'$

$e_2 \rightarrow e_2'$

$e_1 e_2 \rightarrow e_1' e_2$

$e_1 e_2 \rightarrow e_1 e_2'$

Why are these semantics non-deterministic?

Example

• We can apply reduction anywhere in a term
  - $(\lambda x. (\lambda y. y) x ((\lambda z. w) x) \rightarrow \lambda x. (\lambda z. w) x \rightarrow \lambda x. w$
  - $(\lambda x. (\lambda y. y) x ((\lambda z. w) x) \rightarrow \lambda x. (\lambda y. y x (w)) \rightarrow \lambda x. w$

• Does the order of evaluation matter?

The Church-Rosser Theorem

• If $a \rightarrow^{*} b$ and $a \rightarrow^{*} c$, there there exists $d$ such that $b \rightarrow^{*} d$ and $c \rightarrow^{*} d$

• Church-Rosser is also called confluence

Normal Form

• A term is in normal form if it cannot be reduced
  - Examples: $\lambda x. x$, $\lambda x. \lambda y. z$

• By Church-Rosser Theorem, every term reduces to at most one normal form
  - Warning: All of this applies only to the pure lambda calculus with non-deterministic evaluation

• Notice that for our application rule, the argument need not be in normal form
Beta-Equivalence

• Let $\beta$ be the reflexive, symmetric, and transitive closure of $\rightarrow$
  
  E.g., $(\lambda x.x) \ y \rightarrow y \leftarrow (\lambda z.\lambda w.z) \ y \ y$, so all three are beta equivalent

• If $a = \beta b$, then there exists $c$ such that $a \rightarrow^* c$ and $b \rightarrow^* c$
  
  Proof: Consequence of Church-Rosser Theorem

• In particular, if $a = \beta b$ and both are normal forms, then they are equal

A Fixpoint Combinator

• Also called a paradoxical combinator
  
  $Y = \lambda f. (\lambda x. f (x \ x)) \ (\lambda x. f (x \ x))$
  
  Note: There are many versions of this combinator

• Then $Y \ F = \beta F \ (Y \ F)$
  
  $Y \ F = (\lambda f. (\lambda x. f (x \ x)) \ (\lambda x. f (x \ x))) \ F$
  
  $\rightarrow (\lambda x. F (x \ x)) \ (\lambda x. F (x \ x))$
  
  $\rightarrow F ((\lambda x. F (x \ x)) \ (\lambda x. F (x \ x)))$
  
  $\leftarrow F \ (Y \ F)$

Not Every Term Has a Normal Form

• Consider
  
  $\Delta = \lambda x.x \ x$
  
  Then $\Delta \ \Delta \rightarrow \Delta \ \Delta \rightarrow \cdots$

• In general, self application leads to loops
  
  ...which is good if we want recursion

Example

• Fact $n = \text{if } n = 0 \text{ then 1 else } n * \text{fact}(n-1)$

• Let $G = \lambda f. \langle \text{body of factorial} \rangle$

  I.e., $G = \lambda f. \lambda n. \text{if } n = 0 \text{ then 1 else } n^* \text{f}(n-1)$

• $Y \ G \ 1 = \beta G \ (YG) \ 1$
  
  $= \beta (\lambda f. \lambda n. \text{if } n = 0 \text{ then 1 else } n^* \text{f}(n-1)) \ (YG) \ 1$
  
  $= \beta \text{if } 1 = 0 \text{ then 1 else } 1^* ((YG) \ 0)$
  
  $= \beta \text{if } 1 = 0 \text{ then 1 else } 1^* (G \ (YG) \ 0)$
  
  $= \beta \text{if } 1 = 0 \text{ then 1 else } 1^* (\lambda f. \lambda n. \text{if } n = 0 \text{ then 1 else } n^* \text{f}(n-1) \ (YG) \ 0)$
  
  $= \beta \text{if } 1 = 0 \text{ then 1 else } 1^* (\text{if } 0 = 0 \text{ then 1 else } 0^* ((YG) \ 0))$
  
  $= \beta \text{if } 1 = 0 \text{ then 1 else } 1^* = 1$
The Y combinator "unrolls" or "unfolds" its argument an infinite number of times:
- \( Y \ G = G \ (Y \ G) = G \ (G \ (Y \ G)) = \ldots \)
- \( G \) needs to have a "base case" to ensure termination.

But, only works because we're call-by-name:
- Different combinator(s) for call-by-value
  - \( Z = \lambda f. (\lambda x. f (\lambda y. x x y)) \ (\lambda x. f (\lambda y. x x y)) \)
- Why is this a fixed-point combinator? How does its difference from \( Y \) make it work for call-by-value?

In Other Words

• The Y combinator “unrolls” or “unfolds” its argument an infinite number of times
  - \( Y \ G = G \ (Y \ G) = G \ (G \ (Y \ G)) = \ldots \)
  - \( G \) needs to have a “base case” to ensure termination.

• But, only works because we’re call-by-name
  - Different combinator(s) for call-by-value
    - \( Z = \lambda f. (\lambda x. f (\lambda y. x x y)) \ (\lambda x. f (\lambda y. x x y)) \)
    - Why is this a fixed-point combinator? How does its difference from \( Y \) make it work for call-by-value?

Encodings

• Encodings are fun
• They show language expressiveness

• In practice, we usually add constructs as primitives
  - Much more efficient
  - Much easier to perform program analysis on and avoid silly mistakes with
    - E.g., our encodings of true and 0 are exactly the same, but we may want to forbid mixing booleans and integers

Lazy vs. Eager Evaluation

• Our non-deterministic reduction rule is fine for theory, but awkward to implement

• Two deterministic strategies:
  - Lazy: Given \( (\lambda x. e_1) \) \( e_2 \), do not evaluate \( e_2 \) if \( x \) does not “need” \( e_1 \)
    - Also called left-most, call-by-name, call-by-need, applicative, normal-order (with slightly different meanings)
  - Eager: Given \( (\lambda x. e_1) \) \( e_2 \), always evaluate \( e_2 \) fully before applying the function
    - Also called call-by-value

Lazy Operational Semantics

\[
\begin{align*}
(\lambda x. e_1) & \rightarrow^I (\lambda x. e_1) \\
 e_1 & \rightarrow^I \lambda x.e \quad e[e2\langle x \rangle] \rightarrow^I e' \\
 e_1 \ e_2 & \rightarrow^I e'
\end{align*}
\]

• The rules are deterministic and big-step
  - The right-hand side is reduced “all the way”
• The rules do not reduce under \( \lambda \)
• The rules are normalizing:
  - If \( a \) is closed and there is a normal form \( b \) such that \( a \rightarrow^* b \), then \( a \rightarrow^I d \) for some \( d \)
Eager (Big-Step) Op. Semantics

\[(\lambda x.e_1) \rightarrow^e (\lambda x.e_1)\]

\[e_1 \rightarrow^e \lambda x.e \quad e_2 \rightarrow^e e' \quad e'[e'[x] \rightarrow^e e''\]

\[e_1 e_2 \rightarrow^{e''} e''\]

- This big-step semantics is also deterministic and does not reduce under \(\lambda\).
- But it is not normalizing
  - Example: \(\text{let } x = \Delta \Delta \text{ in } (\lambda y.y)\)

Lazy vs. Eager in Practice

- Lazy evaluation (call by name, call by need)
  - Has some nice theoretical properties
  - Terminates more often
  - Lets you play some tricks with “infinite” objects
  - Main example: Haskell

- Eager evaluation (call by value)
  - Is generally easier to implement efficiently
  - Blends more easily with side effects
  - Main examples: Most languages (C, Java, ML, etc.)

Functional Programming

- The \(\lambda\) calculus is a prototypical functional programming language:
  - Lots of higher-order functions
  - No side-effects

- In practice, many functional programming languages are “impure” and permit side-effects
  - But you’re supposed to avoid using them

Functional Programming Today

- Two main camps:
  - Haskell – Pure, lazy functional language; no side effects
  - ML (SML/NJ, OCaml) – Call-by-value, with side effects

- Still around: LISP, Scheme
  - Disadvantage/advantage: No static type systems
Influence of Functional Programming

- Functional ideas in many other languages
  - Garbage collection was first designed with Lisp; most languages often rely on a GC today
  - Generics in Java/C++ came from polymorphism in ML and from type classes in Haskell
  - Higher-order functions and closures (used widely in Ruby; proposed extension to Java) are pervasive in all functional languages
  - Many data abstraction principles of OO came from ML's module system
- ...