CS 476 – Programming Language Design

William Mansky

Questions

Nobody has responded yet.

Hang tight! Responses are coming in.

Functional Programming

- *Functions* are the basic unit of computation
- Functions are values! (“first-class functions”)
  - Functions can take functions as arguments, return functions, etc.
- Immutable variables by default
- Everything is an expression, state changes (“side effects”) are specially marked
- Usually contrasted with imperative languages
- Examples: F#, OCaml, Lisp, Haskell, lambda-expressions

The First Functional Language

- Functional languages are older than computers!
- The *lambda calculus* was invented as a mathematical model of "what can be computed", and it consists entirely of functions

Math notation	Lambda calculus	OCaml
$f(x) = x + 1$	$\lambda x. x + 1$	`fun x -> x + 1`
$g(x, y) = y$	$\lambda x. (\lambda y. y)$	`fun x y -> y`

Multi-Argument Functions

- Math notation: $g(x, y) = x + y$

  $g(1, 2)$ returns 3
- OCaml notation: fun x y -> x + y

  (fun x y -> x + y) 1 2 returns 3
- What about: (fun x y -> x + y) 1

  (fun x y -> x + y) 1 returns (fun y -> 1 + y)

Multi-Argument Functions

- Math notation: $g(x, y) = x + y$

  $g(1, 2)$ returns 3
- OCaml notation: fun x y -> x + y

  (fun x y -> x + y) 1 2 returns 3
- What about: (fun x y -> x + y) 1

  (fun x y -> x + y) 1 returns (fun y -> 1 + y)

  And then (fun y -> 1 + y) 2 returns 3

Multi-Argument Functions

- OCaml notation: fun x y -> x + y

  (fun x y -> x + y) 1 2 returns 3

- What about: (fun x y -> y) 1

  (fun x y -> x + y) 1 returns (fun y -> 1 + y)

  And then (fun y -> 1 + y) 2 returns 3

- (fun x y -> x + y) 1 2 is actually ((fun x -> (fun y -> x + y)) 1) 2

  "Apply this function to 1, get back another function, and then apply that new function to 2"

Multi-Argument Functions

- (fun x y -> x + y) 1 2 is actually ((fun x y -> x + y) 1) 2

  "Apply this function to 1, get back another function, and then apply that new function to 2"

- This is called "currying": to make a function that takes multiple arguments, write a function that returns another function!

- In lambda calculus, we write the same function as $\lambda x.\,(\lambda y.\,x + y)$

Questions

Nobody has responded yet.

Hang tight! Responses are coming in.

Lambda Calculus Basics

- Functions are values, and functions are the only values!
- No declarations, no lets, just anonymous functions
- A function has two parts: $\lambda \underbrace{x}_{\substack{\text{argument name} \\ \text{“bound variable”}}} . \underbrace{B}_{\text{body (any term, can contain } x\text{)}}$

- Functions can be *applied* to other terms (also functions)
- Application is evaluated by replacing the bound variable with the argument in the body

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

Lambda Calculus Basics

- Functions are values, and functions are the only values!
- No declarations, no lets, just anonymous functions
- A function has two parts: $\lambda \underbrace{x}_{\text{argument name “bound variable”}} . \underbrace{B}_{\text{body (any term, can contain } x\text{)}}$

- Functions can be *applied* to other terms (also functions)
- Application is evaluated by replacing the bound variable with the argument in the body

$$(\lambda x.(\lambda y.x))\ z \to (\lambda y.x) \text{ with } x \text{ replaced by } z$$

Lambda Calculus Basics

- Functions are values, and functions are the only values!
- No declarations, no lets, just anonymous functions
- A function has two parts: $\lambda \underbrace{x}_{\text{argument name}} . \underbrace{B}_{\text{body (any term, can contain } x\text{)}}$

argument name
"bound variable"

body (any term, can contain x)

- Functions can be *applied* to other terms (also functions)
- Application is evaluated by replacing the bound variable with the argument in the body

$$(\lambda x. (\lambda y. x))\ z \to (\lambda y. x) \text{ with } x \text{ replaced by } z \text{ i.e., } \lambda y. z$$

Variable Binding

```
int f(int x){ return x + 1; }

int x = 5;
f(x + 2);
```

Lambda Calculus: Binding and Scope

- $\lambda x. B$ binds x in B
- In other words, wherever x appears in B, it means "the argument passed to this function"
- Each variable refers to the *innermost* λ-binding around it

$$\lambda x.\Big(\big(\lambda x. x\ (\lambda x. x\ x)\big)\ x\Big)$$

- A variable that is not bound is *free*, like y in $\lambda x. y\ x$

Lambda Calculus: Renaming

- The name of the argument to a function doesn't really matter
- $\lambda x.\, x$ is the same as $\lambda y.\, y$
- We can always *rename* a bound variable

$$\lambda x.\left(\left(\lambda x.\, x\ (\lambda x.\, x\ x)\right)\ x\right)$$

Lambda Calculus: Renaming

- The name of the argument to a function doesn't really matter
- $\lambda x. x$ is the same as $\lambda y. y$
- We can always *rename* a bound variable

$$\lambda x. \Big(\big(\lambda x. x \; (\lambda y. y \; y) \big) \; x \Big)$$

Lambda Calculus: Renaming

- The name of the argument to a function doesn't really matter
- $\lambda x.x$ is the same as $\lambda y.y$
- We can always *rename* a bound variable

$$\lambda x.\left(\left(\lambda z.z\ (\lambda y.y\ y)\right)\ x\right)$$

- Renaming (sometimes called "alpha-conversion") shouldn't change the behavior of a function

Questions

Nobody has responded yet.

Hang tight! Responses are coming in.

Lambda Calculus: Syntax

L ::= <ident> | λ<ident>. L | L L

- Everything is a function, so there are no interesting types
  - Every function takes a function and returns a function
- The only kind of term that steps to anything is application
  - So the only question for semantics is "how do we apply a function to an argument?"

Lambda Calculus: Substitution

- In general, $(\lambda x.\, l)\ l_2$ evaluates to $[x \mapsto l_2]l$ ("l with l_2 substituted for x")

- $(\lambda x.\, x)\ z$ evaluates to

Lambda Calculus: Substitution

- In general, $(\lambda x. l)\ l_2$ evaluates to $[x \mapsto l_2]l$ ("l with l_2 substituted for x")

- $(\lambda x. x)\ z$ evaluates to z
- Exercise: $\big(\lambda x. (\lambda y. x)\big)\ (\lambda z. z)$ evaluates to $[x \mapsto (\lambda z. z)](\lambda y. x)$ which is $\big(\lambda y. (\lambda z. z)\big)$

Lambda Calculus: Substitution

- In general, $(\lambda x. l)\ l_2$ evaluates to $[x \mapsto l_2]l$ ("l with l_2 substituted for x")

- $(\lambda x. x)\ z$ evaluates to z
- Exercise: $\left(\lambda x. (\lambda y. x)\right)\ z$ evaluates to ?

Lambda Calculus: Substitution

- In general, $(\lambda x. l)\ l_2$ evaluates to $[x \mapsto l_2]l$ ("l with l_2 substituted for x")

- $(\lambda x. x)\ z$ evaluates to z
- $\big(\lambda x. (\lambda y. x)\big)\ (\lambda z. z)$ evaluates to $[x \mapsto (\lambda z. z)](\lambda y. x)$ which is $\big(\lambda y. (\lambda z. z)\big)$

Lambda Calculus: Substitution

- In general, $(\lambda x.\, l)\ l_2$ evaluates to $[x \mapsto l_2]l$ ("l with l_2 substituted for x")

- $(\lambda x.\, x)\ z$ evaluates to z
- $\big(\lambda x.\, (\lambda y.\, x)\big)\ z$ evaluates to $[x \mapsto z](\lambda y.\, x)$ which is $\lambda y.\, z$
- $\big(\lambda x.\, (\lambda y.\, y)\big)\ z$ evaluates to

Lambda Calculus: Substitution

- In general, $(\lambda x.\, l)\ l_2$ evaluates to $[x \mapsto l_2]l$ ("l with l_2 substituted for x")
- $(\lambda x.\, x)\ z$ evaluates to z
- $\left(\lambda x.\, (\lambda y.\, x)\right)\ z$ evaluates to $\lambda y.\, z$
- $\left(\lambda x.\, (\lambda y.\, y)\right)\ z$ evaluates to $\lambda y.\, y$
- $\left(\lambda x.\, \left(x\ (\lambda y.\, y)\right)\right)\ z$ evaluates to

Lambda Calculus: Substitution

- In general, $(\lambda x. l)\ l_2$ evaluates to $[x \mapsto l_2]l$ ("l with l_2 substituted for x")
- $(\lambda x. x)\ z$ evaluates to z
- $(\lambda x. (\lambda y. x))\ z$ evaluates to $\lambda y. z$
- $(\lambda x. (\lambda y. y))\ z$ evaluates to $\lambda y. y$
- $\left(\lambda x. (x\ (\lambda y. y))\right)\ z$ evaluates to $z\ (\lambda y. y)$
- $\left(\lambda x. (x\ (\lambda x. x))\right)\ z$ evaluates to

Lambda Calculus: Substitution

- In general, $(\lambda x.\, l)\ l_2$ evaluates to $[x \mapsto l_2]l$ ("l with l_2 substituted for x")

- $(\lambda x.\, x)\ z$ evaluates to z
- $\big(\lambda x.\, (\lambda y.\, x)\big)\ z$ evaluates to $\lambda y.\, z$
- $\big(\lambda x.\, (\lambda y.\, y)\big)\ z$ evaluates to $\lambda y.\, y$
- $\Big(\lambda x.\, \big(x\ (\lambda y.\, y)\big)\Big)\ z$ evaluates to $z\ (\lambda y.\, y)$
- $\Big(\lambda x.\, \big(x\ (\lambda x.\, x)\big)\Big)\ z$ evaluates to $z\ (\lambda x.\, x)$

Lambda Calculus: Syntax

L ::= <ident> | λ<ident>. L | L L

Lambda Calculus: Semantics

L ::= <ident> | λ<ident>. L | L L

- Functions are values

Lambda Calculus: Semantics

L ::= <ident> | λ<ident>. L | $L\ L$

- $\lambda x. l$ is a value
- Application is evaluated by *substitution*
- $[x \mapsto l_2]l$ means "replace all the x's in l with l_2"

$\left(\lambda y. (\lambda x. x\ y)\right)\ z$ evaluates to $[y \mapsto z](\lambda x. x\ y)$
which is $(\lambda x. x\ z)$

Lambda Calculus: Semantics

L ::= <ident> | λ<ident>. L | L L

- $\lambda x.\, l$ is a value

$$\frac{l_1 \to l_1'}{l_1\ l_2 \to l_1'\ l_2} \qquad\qquad \frac{}{(\lambda x.\, l)\ l_2 \to [x \mapsto l_2]l}$$

- “Call by name”

Lambda Calculus: Semantics

L ::= <ident> | λ<ident>. *L* | *L L*

$$\frac{l_1 \to l_1'}{l_1\ l_2 \to l_1'\ l_2}$$

$$\frac{l_2 \to l_2'}{v\ l_2 \to v\ l_2'}$$

$$\frac{}{(\lambda x.\, l)\ v \to [x \mapsto v]l}$$

- "Call by value"

Call-By-Name vs. Call-By-Value

$(\lambda x.(\lambda y.y))\ l$ where l becomes a value in 10 steps

Call-by-name: "evaluate the arg when it's used"

$\rightarrow (\lambda y.y)$

Call-by-value: "evaluate the arg when it's passed"

$\rightarrow (\lambda x.(\lambda y.y))\ l_1 \rightarrow (\lambda x.(\lambda y.y))\ l_2 \rightarrow \cdots \rightarrow (\lambda x.(\lambda y.y))\ v$

$\rightarrow (\lambda y.y)$

Call-By-Name vs. Call-By-Value

$(\lambda x.(\lambda y.y))\ l$ where l runs forever
$(\lambda x.(x\ x))\ (\lambda x.(x\ x)) \to [x \mapsto (\lambda x.(x\ x))](x\ x)$
which is $(\lambda x.(x\ x))\ (\lambda x.(x\ x))$!

Call-by-name:
$\to (\lambda y.y)$

Call-by-value:
$\to (\lambda x.(\lambda y.y))\ l \to (\lambda x.(\lambda y.y))\ l \to \cdots$

Call-By-Name vs. Call-By-Value

$(\lambda x. \ldots x \ldots\ x \ldots)\ l$ where l becomes a value in 10 steps

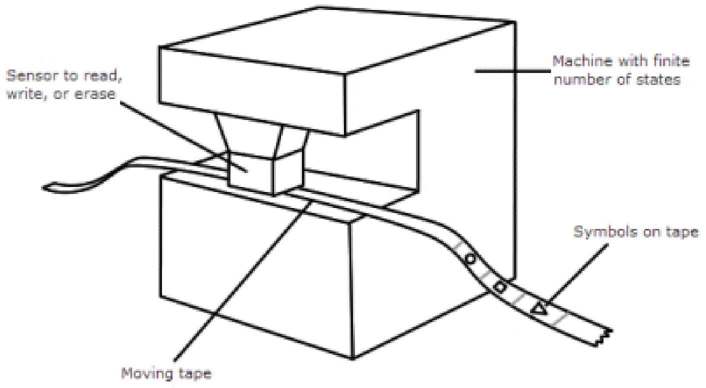
Call-by-name:
$\rightarrow\ \ldots l \ldots l \ldots \rightarrow\ \ldots l_1 \ldots l \ldots \rightarrow\ \ldots v \ldots l \ldots \rightarrow\ \ldots v \ldots l_1 \ldots \rightarrow \cdots$

Call-by-value:
$\rightarrow (\lambda x. \ldots x \ldots x \ldots)\ l_1 \rightarrow \cdots \rightarrow (\lambda x. \ldots x \ldots x \ldots)\ v \rightarrow\ \ldots v \ldots v \ldots$

Lambda Calculus and Computability

What can be computed?

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



Church, 1936

"Turing-complete"

Turing, 1936

Questions

Nobody has responded yet.

Hang tight! Responses are coming in.

Why Functional Programming?

- Lambda calculus has some unusual ideas:
  - Explicit variable binding
  - Evaluation by substitution
  - Minimal shared context between functions
- This is useful for theory:
  - Closer to mathematical functions
  - Very simple semantics
  - Variable binding, scope, etc. is actually the same as in other languages, but lambda calculus lets us see it more directly

Why Functional Programming?

- Lambda calculus has some unusual ideas:
  - Explicit variable binding
  - Evaluation by substitution
  - Minimal shared context between functions
- This is useful for theory
- And in practice!
  - Programs closer to on-paper task descriptions
  - Parallelizes very well (no shared state, mostly pure math)
  - Functions as data is useful (most modern languages have lambdas)