Week 10

Introducing the “lambda calculus”

Phil

Outline

Updates!

Weekly updates:

Updates!

Weekly updates:

- no stickers yet :(

Section 1

Definitions

Models of Computation

- Previously we talked about **Turing machines** and **Turing completeness**

Models of Computation

- Previously we talked about **Turing machines** and **Turing completeness**
- The **Church-Turing thesis**, proven in the 1930s by *Alonzo Church* and *Alan Turing*, states, roughly, that functions on natural numbers are "effectively calculable" iff a Turing machine can compute it.

Models of Computation

- Previously we talked about **Turing machines** and **Turing completeness**
- The **Church-Turing thesis**, proven in the 1930s by *Alonzo Church* and *Alan Turing*, states, roughly, that functions on natural numbers are "effectively calculable" iff a Turing machine can compute it.
- But are there other models of computation that can "effectively calculate" numbers? Then they should be Turing complete, right?

λ Calculus?

Yes! Alonzo Church's *lambda calculus* looks very different from a Turing machine. But it has all of the same capabilities.

$$Y = \lambda f \cdot (\lambda x \cdot \left(f\,(x\,x)\right) \lambda x \cdot (f\,(x\,x)))$$

A short definition

The lambda calculus looks similar to a functional programming language (really, it'd be more accurate to say that they look like the lambda calculus). We'll define a grammar to describe how to parse the language, and a very short list of operations used to "evaluate" lambda calculus expressions.

A short definition

The lambda calculus looks similar to a functional programming language (really, it'd be more accurate to say that they look like the lambda calculus). We'll define a grammar to describe how to parse the language, and a very short list of operations used to "evaluate" lambda calculus expressions.

Remember BNFs from parsing? Here's the lambda calculus BNF:

$$
\begin{aligned}
e ::= {} & x && // \text{ variable} \\
\mid {} & e\ e && // \text{ function application} \\
\mid {} & \lambda x\ .\ e && // \text{ function definition}
\end{aligned}
$$

Short Examples

- **x** is just a variable x.
- **y** is just a variable y. Are these programs equivalent?

Short Examples

- **x** is just a variable x.
- **y** is just a variable y. Are these programs equivalent? (yes)
- **x y** is an application of "x" to "y". Both "x" and "y" are variables.

Short Examples

- **x** is just a variable x.
- **y** is just a variable y. Are these programs equivalent? (yes)
- **x y** is an application of "x" to "y". Both "x" and "y" are variables.
- λ **x . x** defines a function (sometimes called abstraction). It "takes x" and its body is the expression "x". what kind of function is this?

Short Examples

- **x** is just a variable x.
- **y** is just a variable y. Are these programs equivalent? (yes)
- **x y** is an application of "x" to "y". Both "x" and "y" are variables.
- λ **x . x** defines a function (sometimes called abstraction). It "takes x" and its body is the expression "x". what kind of function is this? (looks like the identity function)

Parentheses

- How do we parse $\underline{\lambda\,\mathbf{x}\,.\,\lambda\,\mathbf{y}\,.\,\mathbf{x}\;\mathbf{y}\;\mathbf{x}}$?

Parentheses

- How do we parse $\underline{\lambda\, \mathbf{x}\ .\ \lambda\, \mathbf{y}\ .\ \mathbf{x\ y\ x}}$?
  - Assume function body extends as far right as possible
  - Successive applications are left associative.

Parentheses

- How do we parse $\underline{\lambda\ \mathbf{x}\ .\ \lambda\ \mathbf{y}\ .\ \mathbf{x}\ \mathbf{y}\ \mathbf{x}}$?
  - Assume function body extends as far right as possible
  - Successive applications are left associative.
- Equivalent to $\underline{\lambda\ \mathbf{x}\ .\ (\lambda\ \mathbf{y}\ .\ (\mathbf{x}\ \mathbf{y})\ \mathbf{x})}$

Questions?

Section 2

Semantics

β Reduction

- quick: **value** $::= \lambda x.e$
- Here's the first operation we'll define... it's how to evaluate an expression:

β Reduction

- quick: **value** $::= \lambda x.e$
- Here's the first operation we'll define... it's how to evaluate an expression:
  - $(\lambda\ \mathbf{x}\ .\ \mathbf{e})\ \mathbf{v} \rightarrow \mathbf{e}\ [\ \mathbf{v}\ /\ \mathbf{x}\ ]$
  - in words: if we have a *function* applied to a *value*, then
  - it is "β equivalent" to a *substitution:*
  - where we take "e" and replace all "x" with "v".

β Reduction

- quick: **value** ::= $\lambda x.e$
- Here's the first operation we'll define... it's how to evaluate an expression:
  - $(\lambda\ \mathbf{x}\ .\ \mathbf{e})\ \mathbf{v} \rightarrow \mathbf{e}\ [\ \mathbf{v}\ /\ \mathbf{x}\ ]$
  - in words: if we have a *function* applied to a *value*, then
  - it is "β equivalent" to a *substitution:*
  - where we take "e" and replace all "x" with "v".
- Here's an example:
  - $(\lambda\ \mathrm{x}\ .\ \mathrm{x})\ (\lambda\ \mathrm{y}\ .\ \mathrm{z}) \rightarrow \mathrm{x}\ [\ (\lambda\ \mathrm{y}\ .\ \mathrm{z})\ /\ \mathrm{x}\ ]$

β Reduction

- quick: **value** ::= $\lambda x.e$
- Here's the first operation we'll define... it's how to evaluate an expression:
  - **(λ x . e) v** $\rightarrow$ **e** [**v** / **x**]
  - in words: if we have a *function* applied to a *value*, then
  - it is "β equivalent" to a *substitution:*
  - where we take "e" and replace all "x" with "v".
- Here's an example:
  - (λ x . x) (λ y . z) $\rightarrow$ x [(λ y . z) / x]
  - x [(λ y . z) / x] $\rightarrow$ **λ y . z**

β Reduction Rules

In general, when simplifying an application: β reduce the left side, then the right side *if the left side is a value*, then if both sides are values, β reduce the application.

$$e_1 \;\; e_2 \rightarrow e_1' \;\; e_2 \qquad \text{only if } e_1 \rightarrow e_1'$$

$$v \;\; e_2 \rightarrow v \;\; e_2' \qquad \text{only if } e_2 \rightarrow e_2'$$

Questions?

Now, some linguistics

Let's look at some sentences!

- Susie lost her book.

Now, some linguistics

Let's look at some sentences!

- Susie lost her book.
- Susie lost <u>her</u> book.

Now, some linguistics

Let's look at some sentences!

- Susie lost her book.
- Susie lost <u>her</u> book.
- Susie lost <u>her</u> book. *Alice had a book.* (whose book did Susie lose?)

Now, some linguistics

Let's look at some sentences!

- Susie lost her book.
- Susie lost her book.
- Susie lost her book. *Alice had a book.* (whose book did Susie lose?)
- Susie lost her own book. (oh, Susie lost her own book)

Free and Bound Variables

- What is "y" here? $\underline{\lambda\ \mathbf{x}\ .\ \lambda\ \mathbf{y}\ .\ \mathbf{y}}$
- What is "x" here? $\underline{\lambda\ \mathbf{x}\ .\ \lambda\ \mathbf{y}\ .\ \mathbf{x}}$

Free and Bound Variables

- What is "y" here? $\underline{\lambda\ \mathbf{x}\ .\ \lambda\ \mathbf{y}\ .\ \mathbf{y}}$
- What is "x" here? $\underline{\lambda\ \mathbf{x}\ .\ \lambda\ \mathbf{y}\ .\ \mathbf{x}}$
- **Bound variables** refer to the argument of the function they are in.
- **Free variables** refer to a variable declared outside the function body.

α Renaming

Let's say we have $(\lambda$ x . y) and want to substitute y with "x". How do we do that?

α Renaming

Let's say we have $(\lambda$ x . y) and want to substitute y with "x". How do we do that?

- What kind of variable is y in $(\lambda$ x . y) ?

α Renaming

Let's say we have $(\lambda$ x . y$)$ and want to substitute y with "x". How do we do that?

- What kind of variable is y in $(\lambda$ x . y$)$? (free)
- When we are substituting in new values for free variables, we have to avoid naming conflicts.
- α equivalent - we can rename variables as long as it doesn't change the meaning of the program.

We change the expression to $(\lambda$ z . y$)$ and then substitute "x" for "y" to get $(\lambda$ z . x$)$. Great!

Exercise

Let's try a more complex one, try reducing this expression (finding its normal form):

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

Exercise

Let's try a more complex one, try reducing this expression (finding its normal form):

- $((\lambda\ \text{x}.\ \lambda\ \text{y}.\ \text{x y})\ (\lambda\ \text{y}\ .\ \text{y}))\ (\lambda\ \text{z}\ .\ \text{z})$
- $(\lambda\ \text{w}.\ (\lambda\ \text{y}\ .\ \text{y})\ \text{w})\ (\lambda\ \text{z}\ .\ \text{z})$
- $(\lambda\ \text{y}\ .\ \text{y})\ (\lambda\ \text{z}\ .\ \text{z})$
- $(\lambda\ \text{z}\ .\ \text{z})$

Section 3

Abstract Data Types

Computation?

How do we actually compute in lambda calculus?

Computation?

How do we actually compute in lambda calculus?
Well, we have to create some **abstract data types**. We'll define what a data type is in lambda calculus, and then define functions that return expressions matching our data types.

Computation?

How do we actually compute in lambda calculus?
Well, we have to create some **abstract data types**. We'll define what a data type is in lambda calculus, and then define functions that return expressions matching our data types.

Let's define booleans:

- True ::= $\lambda x.\lambda y.x$
- False ::= $\lambda x.\lambda y.y$

Can you write a lambda calculus function defining the conditional? (a function, given a boolean, P, and Q, returns either P or Q)

What about OR (a function, given two booleans, returns the OR of the two) (you can use "true" or "false" in your answer)

Computation Answers

Conditional:

Computation Answers

Conditional:

- $\lambda c. \lambda p. \lambda q.\ c\ p\ q$

Computation Answers

Conditional:

- $\lambda c. \lambda p. \lambda q.\ c\ p\ q$
- explanation: our boolean "takes in" two arguments. if its true, it returns the first input, if its false, it returns the second, by definition.

OR gate:

Computation Answers

Conditional:

- $\lambda c. \lambda p. \lambda q.\ c\ p\ q$
- explanation: our boolean "takes in" two arguments. if its true, it returns the first input, if its false, it returns the second, by definition.

OR gate:

- $\lambda a. \lambda b.\ a \text{ true } b$
- this should remind you of *short circuit evaluation* if you've learned that about programming languages. if A, then true, else return B.

Section 4

Y Combinator

Something's missing...

Does lambda calculus support recursion?

Something's missing...

Does lambda calculus support recursion?
Let's look at the Y combinator: $Y = \lambda f((\lambda x.f(x\ x))(\lambda x.f(x\ x)))$.
What if we apply a function g to this?

$Y\ g =$

- Substituting g for f:
- $Y\ g = (\lambda x.\ g\ (x\ x))\ (\lambda x.\ g\ (x\ x))$

Something's missing...

Does lambda calculus support recursion?
Let's look at the Y combinator: $Y = \lambda f((\lambda x.f(x\ x))(\lambda x.f(x\ x)))$.
What if we apply a function g to this?

$Y\ g =$

- Substituting g for f:
- $Y\ g = (\lambda x.\ g\ (x\ x))\ (\lambda x.\ g\ (x\ x))$
- Applying:
- $Y\ g = g\ ((\lambda x.\ g\ (x\ x))\ (\lambda x.\ g\ (x\ x)))$
- $Y\ g = g\ (Y\ g)$

Y Combinator

What is a combinator?

- a "higher-order" function that returns a fixed point of its argument function.
- "higher-order" means it takes in a function as an argument. For example a sort function can take in a function telling you how to order the elements.
- fixed point of a function f (call it fix f) means fix f = f(fix f) = f(f(fix f)) = ...

Y Combinator

What is a combinator?

- a "higher-order" function that returns a fixed point of its argument function.
- "higher-order" means it takes in a function as an argument. For example a sort function can take in a function telling you how to order the elements.
- fixed point of a function f (call it fix f) means fix f = f(fix f) = f(f(fix f)) = ...

Y Combinator is just one example. There are actually infinitely many combinator functions possible in the lambda calculus.

Y Combinator Useful for?

Remember, you **can't** refer to a function within the function in lambda calculus (put differently, all functions are anonymous, unnameable to themselves)

Y Combinator Useful for?

Remember, you **can't** refer to a function within the function in lambda calculus (put differently, all functions are anonymous, unnameable to themselves)

But you **can** use the Y combinator to pass a function to itself... thereby always giving it a reference with which to call itself!

Y Combinator Useful for?

Remember, you **can't** refer to a function within the function in lambda calculus (put differently, all functions are anonymous, unnameable to themselves)

But you **can** use the Y combinator to pass a function to itself... thereby always giving it a reference with which to call itself!

Example: factorial = λ f . λ n . cond (is_zero n) (1) (mul n (f (pred n))

Y Combinator Useful for?

Remember, you **can't** refer to a function within the function in lambda calculus (put differently, all functions are anonymous, unnameable to themselves)

But you **can** use the Y combinator to pass a function to itself... thereby always giving it a reference with which to call itself!

Example: factorial = λ f . λ n . cond (is_zero n) (1) (mul n (f (pred n))
Key point: Y factorial 2 = factorial (Y factorial) 2 (!!)

Next time...

We toast the Lisp programmer who pens his thoughts within nests of parentheses.
— ALAN PERLIS (1984)