

# Lambda Calculus

It turns out abstraction is pretty powerful

# Lambda Calculus

It turns out **abstraction** is pretty powerful

# abstraction (noun)

removing specific details from something so its concepts can be applied to a broader variety of circumstances

# Example of Abstraction

Recipe for pineapple upside-down cake:

- Put pineapple in the bottom of the cake pan
- Pour the cake batter on top
- Bake it
- Turn it out upside-down so that the pineapple is on top

# Example of Abstraction

Recipe for **pineapple** upside-down cake:

- Put **pineapple** in the bottom of the cake pan
- Pour the cake batter on top
- Bake it
- Turn it out upside-down so that the **pineapple** is on top

# Example of Abstraction

Recipe for **plum** upside-down cake:

- Put **plum** in the bottom of the cake pan
- Pour the cake batter on top
- Bake it
- Turn it out upside-down so that the **plum** is on top

# Example of Abstraction

Recipe for **blueberry** upside-down cake:

- Put **blueberry** in the bottom of the cake pan
- Pour the cake batter on top
- Bake it
- Turn it out upside-down so that the **blueberry** is on top

# Example of Abstraction

An abstraction: adding a hole which can be filled in

Recipe for  upside-down cake:

- Put  in the bottom of the cake pan
- Pour the cake batter on top
- Bake it
- Turn it out upside-down so that the  is on top

Let's **name** this abstraction “**upside-down cake**”



# Example of Abstraction

Applying the abstraction: filling in the hole

- “**pineapple** upside-down cake”
- “**plum** upside-down cake”
- “**blueberry** upside-down cake”

So then what is an abstraction?

- Something with **holes** in it which can be **filled in later**.
- **Filling in the holes** is called **applying the abstraction**.

# When is an abstraction useful?

When it expresses a **concept**  
that is **general enough**  
for there to be **many occasions**  
to **apply**  
the **abstraction**

# Lambda Calculus

A formalization of **abstractions** and **applications**

# Representing Abstraction

- Is the “hole” representation sufficiently precise?
- No; example:

“ are ”

What should be the result of applying this abstraction to “functions”?

- “functions are functions”?
- “functions are ”?
- “ are functions”?
- “ are ”?

# Representing Abstraction

- Solution: to make an abstraction,
  - Replace the hole(s) an abstraction refers to with a **variable**
  - Say which **variable** the abstraction **refers to**
- Let's also use some arbitrary particular symbol to indicate that we're making an abstraction, just to make parsing easier.

“□ are □”

# Representing Application

- Just put the abstraction next to the thing you want to apply it to.
- In particular: abstraction on the left, thing on the right

$((\lambda x. (\lambda y. \text{“}x \text{ are } y\text{”})) \text{functions}) \text{values}$

- Is that all we need to formalize about this calculus?
- We want these expressions to be “equal” in some sense:

$((\lambda x. (\lambda y. \text{“}x \text{ are } y\text{”})) \text{functions}) \text{values} \equiv \text{“functions are values”}$

So we still need to formalize this notion of “equality”

# Specifying What We Want to be “Equal”

- There are a lot of subtly different ways to do this
- I’m going to do what I consider the most satisfying approach, from a PL theory perspective:
- Defining a small-step dynamics for lambda calculus, and expressing equality in terms of it
  - I’ll actually discuss a few different ways to define the dynamics



# The Core of the Dynamics

There are a few rules that people find so interesting that there are names for them:

$$\frac{}{\lambda x.e \rightarrow \lambda y.[y/x]e} \alpha$$

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

$$\frac{}{\lambda x.e x \rightarrow e} \eta$$

# The Core of the Dynamics

I don't find  $\alpha$  or  $\eta$  particularly interesting

$$\overline{\lambda x.e \rightarrow \lambda y.[y/x]e} \quad \alpha$$

$$\overline{(\lambda x.e_1) e_2 \rightarrow [e_2/x]e_1} \quad \beta$$

$$\overline{\lambda x.e \ x \rightarrow e} \quad \eta$$

# Completing the Dynamics: Lazy, Deterministic

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

Consider evaluating this expression if we only have the  $\beta$  rule:

$((\lambda x. (\lambda y. \text{"x are y"}))$  **functions**) **values**

Problem: this expression **can't step** because the expression in the function position **isn't a lambda**

# Completing the Dynamics: Lazy, Deterministic

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

Solution:

$$\frac{e_1 \rightarrow e'_1}{e_1 e_2 \rightarrow e'_1 e_2}$$

# Completing the Dynamics: Lazy, Deterministic

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

$$\frac{e_1 \rightarrow e'_1}{e_1 e_2 \rightarrow e'_1 e_2}$$

# Completing the Dynamics: More Traditional

$$\frac{}{\lambda x.e \rightarrow \lambda y.[y/x]e} \alpha$$

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

$$\frac{}{\lambda x.e \ x \rightarrow e} \eta$$

$$\frac{e_1 \rightarrow e'_1}{e_1 e_2 \rightarrow e'_1 e_2}$$

$$\frac{e_2 \rightarrow e'_2}{e_1 e_2 \rightarrow e_1 e'_2}$$

$$\frac{e \rightarrow e'}{\lambda x.e \rightarrow \lambda x.e'}$$

# Defining Equivalence using Dynamics

$$\frac{e_1 \rightarrow e_2}{e_1 \equiv e_2}$$

$$\frac{}{e \equiv e} \quad \frac{e_1 \equiv e_2}{e_2 \equiv e_1} \quad \frac{e_1 \equiv e_2 \quad e_2 \equiv e_3}{e_1 \equiv e_3}$$

# Definability

Lambda calculus supports every feature you've seen in programming languages




# Definability

- Features of Lambda++ which we'll express in lambda calculus:
  - Tuples
  - Sums
  - Fixed points (what?)
- The key to defining data structures in lambda calculus:  
Asking **how** those data structures are **used**  
A lot of the time it's just a matter of **continuation-passing style** and **currying**

# Definability: Tuples

How is a tuple used?

`let (x, y) = e1 in e2`



So we need the tuple “usage” form to fill in the holes in  $e_2$  with the elements of the tuple

So this will appear somewhere in the “usage” form for tuples:

$\lambda x. \lambda y. e_2$

And it'll need to get applied to the elements of the tuple

# Definability: Tuples

## Definability: Tuples

$$(e_1, e_2) \triangleq \lambda f. f e_1 e_2$$

## Definability: Tuples

$$(e_1, e_2) \triangleq \lambda f. f e_1 e_2$$

$$\mathbf{let} \ (x, y) = e_1 \ \mathbf{in} \ e_2 \triangleq e_1 (\lambda x. \lambda y. e_2)$$

## Definability: Tuples

$$(e_1, e_2) \triangleq \lambda f. f e_1 e_2$$

$$\mathbf{let} \ (x, y) = e_1 \ \mathbf{in} \ e_2 \triangleq e_1 (\lambda x. \lambda y. e_2)$$

$$\#1 \ e \triangleq e (\lambda x. \lambda y. x)$$

## Definability: Tuples

$$(e_1, e_2) \triangleq \lambda f. f e_1 e_2$$

$$\mathbf{let} \ (x, y) = e_1 \ \mathbf{in} \ e_2 \triangleq e_1 (\lambda x. \lambda y. e_2)$$

$$\#1 \ e \triangleq e(\lambda x. \lambda y. x)$$

$$\#2 \ e \triangleq e(\lambda x. \lambda y. y)$$

# Definability: Sums

How is a sum injection used?

case  $e$  of

INL  $x_1 \Rightarrow e_1$

INR  $x_2 \Rightarrow e_2$

So we need the sum “usage” form to select one of the branches and fill in the corresponding hole

So these will appear somewhere in the usage form, and one of them will need to be applied:

$\lambda x_1. e_1$        $\lambda x_2. e_2$



# Definability: Sums

# Definability: Sums

$$\text{INL } e \triangleq \lambda k_1. \lambda k_2. k_1 e$$

# Definability: Sums

$$\text{INL } e \triangleq \lambda k_1. \lambda k_2. k_1 e$$

$$\text{INR } e \triangleq \lambda k_1. \lambda k_2. k_2 e$$

# Definability: Sums

$$\text{INL } e \triangleq \lambda k_1. \lambda k_2. k_1 e$$

$$\text{INR } e \triangleq \lambda k_1. \lambda k_2. k_2 e$$

$$\text{case } e \text{ of } \text{INL } x_1 \Rightarrow e_1 \mid \text{INR } x_2 \Rightarrow e_2 \triangleq e(\lambda x_1. e_1)(\lambda x_2. e_2)$$

# Definability: Fixed Points

First of all, what is a fixed point?

---

$$\text{fix } x \text{ is } e \rightarrow [(\text{fix } x \text{ is } e)/x]e$$

For example:


```
fix fact is
```

```
  fn 0 => 1
```

```
  | n => n * fact (n - 1)
```

# Definability: Fixed Points

`fix x is e`



So we'll give

$\lambda x.e$

to whatever we use to achieve fixed points. Let's call it  $Y$ .

So we want

$$Y(\lambda x.e) \equiv (\lambda x.e)(Y(\lambda x.e))$$

# Definability: Fixed Points

$$Y(F) \equiv F(Y(F))$$

Claim: If we let

$$Y(F) = (\lambda x.F(x\ x))(\lambda x.F(x\ x))$$

then this equivalence will hold.

# Definability: Fixed Points

Goal:  $Y(F) \equiv F(Y(F))$

$$Y(F) = (\lambda x.F(x\ x))(\lambda x.F(x\ x))$$



# Definability: Fixed Points

Goal:  $Y(F) \equiv F(Y(F))$

$$\begin{aligned} Y(F) &= (\lambda x. F(x\ x))(\lambda x. F(x\ x)) \\ &\rightarrow F(\underbrace{((\lambda x. F(x\ x)) (\lambda x. F(x\ x)))}_{Y(F)}) \end{aligned}$$

# Definability: Fixed Points

Goal:  $Y(F) \equiv F(Y(F))$

$$\begin{aligned} Y(F) &= (\lambda x.F(x\ x))(\lambda x.F(x\ x)) \\ &\rightarrow F((\lambda x.F(x\ x))(\lambda x.F(x\ x))) \\ &= F(Y(F)) \end{aligned}$$

# Definability: Fixed Points

Goal:  $Y(F) \equiv F(Y(F))$

$$Y(F) = (\lambda x.F(x\ x))(\lambda x.F(x\ x))$$

# Definability: Fixed Points

Goal:  $Y(F) \equiv F(Y(F))$

$$Y(F) = (\lambda x.F(x\ x))(\lambda x.F(x\ x))$$

$$Y = \lambda F.(\lambda x.F(x\ x))(\lambda x.F(x\ x))$$

# Definability: Fixed Points

$\mathbf{fix} \ x \ \mathbf{is} \ e \triangleq Y(\lambda x.e)$

where  $Y = \lambda F.(\lambda x.F(x \ x))(\lambda x.F(x \ x))$