Lambda Calculus

It turns out abstraction is pretty powerful

Lambda Calculus

It turns out abstraction is pretty powerful

What is Abstraction?

What is Abstraction?

Huh? What is it?

What is Abstraction?

What on earth is **abstraction**?

What is Abstraction?

Anyone else wondering what **abstraction** is?

Example of Abstraction

- "I repeatedly put one of my feet in front of the other until I reached WEH 5421"
- "I repeatedly put one of my feet in front of the other until I reached Fuku Tea"
- "I repeatedly put one of my feet in front of the other until I reached Canada"

Example of Abstraction

- "I repeatedly put one of my feet in front of the other until I reached WEH 5421"
- "I repeatedly put one of my feet in front of the other until I reached Fuku Tea"
- "I repeatedly put one of my feet in front of the other until I reached Canada"

Example of Abstraction

An abstraction: adding a hole which can be filled in

"I repeatedly put one of my feet in front of the other until I reached ________"

Let's **name** this abstraction "**I walked to**"

Example of Abstraction

Applying the abstraction: filling in the hole

- “I walked to WEH 5421”
- “I walked to Fuku Tea”
- “I walked to Canada”

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

- Is the “putting the abstraction to the left of the thing we’re applying it to” representation sufficiently precise?
- Ye

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

((λx. (λy. “x are y”)) functions) values ≡ “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 α or η particularly interesting

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

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

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

Completing the Dynamics: Lazy, Deterministic

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

Consider evaluating this expression if we only have the β rule:

((λx. (λy. "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

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

$$\frac{e_1 \to e_1'}{e_1\ e_2 \to e_1'\ e_2}$$

Completing the Dynamics: More Traditional

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

$$\frac{e_1 \to e_1'}{e_1\ e_2 \to e_1'\ e_2}$$

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

$$\frac{e_2 \to e_2'}{e_1\ e_2 \to e_1\ e_2'}$$

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

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

Defining Equivalence using Dynamics

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

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

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?

$$\texttt{let}\ (x, y) = e_1\ \texttt{in}\ e_2$$

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

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

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

Definability: Tuples

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

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

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

Definability: Tuples

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

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

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

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

Definability: Sums

How is a sum injection used?

$$\texttt{case}\ e\ \texttt{of}$$

$$\texttt{INL}\ x_1 \Rightarrow e_1$$

$$\texttt{INR}\ x_2 \Rightarrow e_2$$

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:

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

Definability: Sums

Definability: Sums

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

Definability: Sums

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

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

Definability: Sums

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

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

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

Definability: Fixed Points

First of all, what is a fixed point?

$$\frac{}{\texttt{fix}\ x\ \texttt{is}\ e \to [(\ \texttt{fix}\ x\ \texttt{is}\ e)/x]e}$$

For example:

```
fix fact is
  fn 0 => 1
   | n => n * fact (n – 1)
```

Definability: Fixed Points

$$\texttt{fix}\ x\ \texttt{is}\ e$$

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

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

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

$$\texttt{fix}\ x\ \texttt{is}\ e \triangleq Y(\lambda x.e)$$
$$\text{where } Y = \lambda F.(\lambda x.F(x\ x))(\lambda x.F(x\ x))$$