Introduction
λ-calculus
Some Functions
Functional Programming

λ-calculus and Functional Programming

Noah Singer

Montgomery Blair High School Computer Team

January 6, 2017

Motivation

Let's consider $f(x) = x^2$ and $g(x) = \sin x$.

Motivation

Let's consider $f(x) = x^2$ and $g(x) = \sin x$.

We can write the **composition** of f and g as $h(x) = f(g(x)) = \sin^2 x$. For example, $h(\frac{\pi}{6}) = \frac{1}{4}$.

Motivation

Let's consider $f(x) = x^2$ and $g(x) = \sin x$.

We can write the **composition** of f and g as $h(x) = f(g(x)) = \sin^2 x$. For example, $h(\frac{\pi}{6}) = \frac{1}{4}$.

What about $h = f(g)$?

We'd have $(f(g))(\frac{\pi}{6})$. Is this meaningful?

Motivation

Passing functions as data around like this is called using **first-class functions**.

Let's consider the function $s(x, y) = x^2 + y^2$. For example, $s(3, 5) = 3^2 + 5^2 = 34$.

Motivation

Passing functions as data around like this is called using **first-class functions**.

Let's consider the function $s(x, y) = x^2 + y^2$. For example, $s(3, 5) = 3^2 + 5^2 = 34$.

We can write this function **anonymously** as $(x, y) \mapsto x^2 + y^2$.

What advantages does this have?

Currying

First, let's restrict ourselves to functions that take one argument.

Currying

First, let's restrict ourselves to functions that take one argument.

How can we accommodate functions that take multiple arguments?

Currying

First, let's restrict ourselves to functions that take one argument.

How can we accommodate functions that take multiple arguments?

Through **currying**: $x \mapsto (y \mapsto x^2 + y^2)$.

Currying

First, let's restrict ourselves to functions that take one argument.

How can we accommodate functions that take multiple arguments?

Through **currying**: $x \mapsto (y \mapsto x^2 + y^2)$.

One of the first cool things we can do is to do **partial application**:

$$[(x \mapsto (y \mapsto x^2 + y^2))(3)](5) = (y \mapsto 9 + y^2)(5) = 34$$

.

The λ-operator

What if we make everything a function? Then we get λ-calculus.

Let's rewrite our anonymous function $x \mapsto (y \mapsto x^2 + y^2)$ as $\lambda x.(\lambda y.(x^2 + y^2))$.

Our function application would be written as:

$$\begin{aligned} ((\lambda x.(\lambda y.(x^2 + y^2))\ 3)\ 5) &= ((\lambda y.(3^2 + y^2))\ 5) \\ &= ((\lambda y.(9 + y^2))\ 5) \\ &= 9 + 5^2 \\ &= 34 \end{aligned}$$

λ-expressions

These three rules together enumerate all the set of all valid **λ-expressions** (or λ-terms), Λ, for some variables V:

1. **Variables**: $x \in V \to x \in \Lambda$. For example, $x \in \Lambda$.
2. **λ-abstraction**: $x \in V, t \in \Lambda \to (\lambda x.t) \in \Lambda$. For example, for $x = x$ and $t = x^2 + 2$, then $(\lambda x.x^2 + 2) \in \Lambda$.
3. **λ-application**: $t, s \in \Lambda \to (ts) \in \Lambda$. For example, if $t = (\lambda x.x^2 + 2)$ and $s = 3$, then $((\lambda x.x^2 + 2)\ 3) \in \Lambda$.

Variables

Variables are either **free** or **bound**, with free variables $F(q)$ of all expressions $q \in \Lambda$ defined as follows:

1. $x \in V \to F(x) = \{x\}$
2. $x \in V, t \in \Lambda \to F(\lambda x.t) = V(t) - \{x\}$
3. $t, s \in \Lambda \to F(ts) = F(t) \cup V(s)$

For example, in $(\lambda x.x + y)$, x is bound while y is free.

λ-terms are very much like functions, but they take functions as input!

α-substitution

Definition (α-substitution)

Replacement in some expression A of one variable by another variable to yield the expression B when it doesn't change the meaning of the expression, denoted $A \underset{\alpha}{\rightarrow} B$.

For example,

- $\lambda x.x \equiv \lambda y.y \equiv \lambda z.z$
- $\lambda x.x \not\equiv \lambda x.y$

Notation

To simplify, for notational purposes, we write the following:

- $\lambda x_0 x_1 x_2 \ldots x_n.t$ means $(\lambda x_0.(\lambda x_1.(\lambda x_2 \ldots (\lambda x_n.t)) \ldots)$
- $E_0 E_1 E_2 \ldots E_n$ means $(\ldots((E_0 E_1) E_2) \ldots) E_n$

More operations

We can **substitute** some variable x for some variable y in t, denoted $t[x := y]$, by replacing all occurrences of x with y. Subject to some annoying restrictions (that we won't list here):

Definition (β-reduction)

The "application" of a λ-term to another λ-term: $(\lambda x.t)s \underset{\beta}{\rightarrow} t[x := s]$.

Definition (η-conversion)

The "abstraction" of a λ-term f: $f \underset{\eta}{\rightarrow} \lambda x.fx$, when x does not appear free in f.

This captures the intuitive notion of function application. We have now defined every possible computer program.

λ-calculus and Functional Programming

Noah Singer

Introduction

λ-calculus

Some Functions

Functional Programming

Wait, what??? How!

Let's find out!

Basic functions

We begin with basic functions.

- *Identity*: $\mathbf{I} \equiv \lambda x.x$
- *Constant*: $\mathbf{C} \equiv \lambda x.y$

Boolean logic functions

Intuitively, we want to have Boolean values **T** and **F** where for some values x and y and a Boolean value B, Bxy evaluates to the expression "if B then x else y". We may accomplish this as follows:

Boolean logic functions

Intuitively, we want to have Boolean values **T** and **F** where for some values x and y and a Boolean value B, Bxy evaluates to the expression "if B then x else y". We may accomplish this as follows:

- *True*: $\mathbf{T} \equiv \lambda xy.x$

  ```
  def T(x,y): return x
  ```

- *False*: $\mathbf{F} \equiv \lambda xy.y$

  ```
  def F(x,y): return y
  ```

Boolean logic functions

Intuitively, we want to have Boolean values **T** and **F** where for some values x and y and a Boolean value B, Bxy evaluates to the expression "if B then x else y". We may accomplish this as follows:

- *True*: $\mathbf{T} \equiv \lambda xy.x$
```
def T(x,y): return x
```
- *False*: $\mathbf{F} \equiv \lambda xy.y$
```
def F(x,y): return y
```
- *Negation*: $\neg \equiv \lambda x.x\mathbf{FT}$ (read: "if x then **F** else **T**")
```
def not(x): return x(F, T)
```

Boolean logic functions

Intuitively, we want to have Boolean values **T** and **F** where for some values x and y and a Boolean value B, Bxy evaluates to the expression "if B then x else y". We may accomplish this as follows:

- *True*: $\mathbf{T} \equiv \lambda xy.x$
```
def T(x,y): return x
```
- *False*: $\mathbf{F} \equiv \lambda xy.y$
```
def F(x,y): return y
```
- *Negation*: $\neg \equiv \lambda x.x\mathbf{FT}$ (read: "if x then **F** else **T**")
```
def not(x): return x(F, T)
```
- *Conjunction*: $\wedge \equiv \lambda xy.xy\mathbf{F}$ (read: "if x then y else **F**")
```
def and(x,y): return x(y, F)
```

Boolean logic functions

Intuitively, we want to have Boolean values **T** and **F** where for some values x and y and a Boolean value B, Bxy evaluates to the expression "if B then x else y". We may accomplish this as follows:

- *True*: $\mathbf{T} \equiv \lambda xy.x$
  ```
  def T(x,y): return x
  ```
- *False*: $\mathbf{F} \equiv \lambda xy.y$
  ```
  def F(x,y): return y
  ```
- *Negation*: $\neg \equiv \lambda x.x\mathbf{FT}$ (read: "if x then **F** else **T**")
  ```
  def not(x): return x(F, T)
  ```
- *Conjunction*: $\wedge \equiv \lambda xy.xy\mathbf{F}$ (read: "if x then y else **F**")
  ```
  def and(x,y): return x(y, F)
  ```
- *Disjunction*: $\vee \equiv \lambda xy.x\mathbf{T}y$ (read: "if x then **T** else y")
  ```
  def or(x,y): return x(T, y)
  ```

Arithmetic

We can also define the natural numbers using λ-calculus, the basic idea being that $\mathbf{x}$, the λ-term corresponding to the natural number x, when applied to a function f and a value v, should apply f x times to v. These are called the **Church numerals**.

- *Zero*: $\mathbf{0} \equiv \lambda fv.v$

Arithmetic

We can also define the natural numbers using λ-calculus, the basic idea being that $\mathbf{x}$, the λ-term corresponding to the natural number x, when applied to a function f and a value v, should apply f x times to v. These are called the **Church numerals**.

- *Zero*: $\mathbf{0} \equiv \lambda fv.v$
- *Natural numbers*: $\mathbf{1} \equiv \lambda fv.f(v), \mathbf{2} \equiv \lambda fv.f(f(v)), \mathbf{3} \equiv \lambda fv.f(f(f(v))), \ldots, \mathbf{x} \equiv \lambda fv.f^x(v)$

Arithmetic

We can also define the natural numbers using λ-calculus, the basic idea being that $\mathbf{x}$, the λ-term corresponding to the natural number x, when applied to a function f and a value v, should apply f x times to v. These are called the **Church numerals**.

- *Zero*: $\mathbf{0} \equiv \lambda fv.v$
- *Natural numbers*: $\mathbf{1} \equiv \lambda fv.f(v), \mathbf{2} \equiv \lambda fv.f(f(v)), \mathbf{3} \equiv \lambda fv.f(f(f(v))), \ldots, \mathbf{x} \equiv \lambda fv.f^{x}(v)$
- *Successor*: $\mathbf{S} \equiv \lambda wfv.f(wfv)$

An example!

For example,

$$
\begin{aligned}
\mathbf{S1} &\equiv (\lambda wfv.y(wfv))(\lambda fv.f(v)) \\
&= \lambda fv.f((\lambda fv.f(v))fv) \\
&= \lambda fv.f(f(v)) \\
&\equiv \mathbf{2}
\end{aligned}
$$

Arithmetic operations

- *Addition*: $+ \equiv \lambda xy.x\mathbf{S}y$

Arithmetic operations

- *Addition*: $+ \equiv \lambda xy.x\mathbf{S}y$
- *Multiplication*: $* \equiv \lambda xyfv.x(yf)v$

Arithmetic operations

- *Addition*: $+ \equiv \lambda xy.x\mathbf{S}y$
- *Multiplication*: $* \equiv \lambda xyfv.x(yf)v$
- *Zero?*: $\mathbf{Z} \equiv \lambda x.x(\lambda y.\mathbf{F})\mathbf{T}$

Recursion

We can't write recursion directly. However, we can still do things recursively by passing functions themselves as arguments: $(\lambda x.xx)y$ calls y on itself.

We define the **Y-combinator** (a **fixed-point combinator**) as follows:

$$\mathbf{Y} \equiv \lambda y.(\lambda x.y(xx))(\lambda x.y(xx))$$

This has the important property that for any R,

$$\begin{aligned}\mathbf{Y}R &\equiv (\lambda x.R(xx))(\lambda x.R(xx)) \\ &= R((\lambda x.R(xx))(\lambda x.R(xx))) \\ &= R(\mathbf{Y}R)\end{aligned}$$

Thus, we have recursion!

Recursion example

We can recursively evaluate functions like the factorial by defining terms like $! = \lambda rn.(\mathbf{Z}n)\mathbf{1}(*n(r(\mathbf{P}n)))$ and then "calling them on themselves".

```
def fact(f, n):
    if n == 0: return 1
    else: return n * f(n-1)
```

$$
\begin{aligned}
(\mathbf{Y}!)\mathbf{3} &\equiv !(\mathbf{Y}!)\mathbf{3} \\
&= (\lambda rn.(\mathbf{Z}n)\mathbf{1}(*n(r(\mathbf{P}n))))(\mathbf{Y}!)\mathbf{3} \\
&= (\lambda n.(\mathbf{Z}n)\mathbf{1}(*n((\mathbf{Y}!)(\mathbf{P}n))))\mathbf{3} \\
&= (\mathbf{Z3})\mathbf{1}(*\mathbf{3}((\mathbf{Y}!)(\mathbf{P3}))) \\
&= *\mathbf{3}((\mathbf{Y}!)\mathbf{2}) \\
&= *\mathbf{3}((\lambda rn.(\mathbf{Z}n)\mathbf{1}(*n(r(\mathbf{P}n))))(\mathbf{Y}!)\mathbf{2}) \\
&= *\mathbf{3}((\lambda n.(\mathbf{Z}n)\mathbf{1}(*n((\mathbf{Y}!)(\mathbf{P}n))))\mathbf{2})
\end{aligned}
$$

Example, continued

$$
\begin{aligned}
&= *\mathbf{3}((\mathbf{Z}\mathbf{2})\mathbf{1}(*\mathbf{2}((\mathbf{Y}!)(\mathbf{P}\mathbf{2}))))\\
&= *\mathbf{3}(*\mathbf{2}((\mathbf{Y}!)\mathbf{1}))\\
&= *\mathbf{3}(*\mathbf{2}((\mathbf{Y}!)\mathbf{1}))\\
&= *\mathbf{3}(*\mathbf{2}(\lambda rn.(\mathbf{Z}n)\mathbf{1}(*n(r(\mathbf{P}n))))(\mathbf{Y}!\mathbf{1}))\\
&= *\mathbf{3}(*\mathbf{2}((\lambda n.(\mathbf{Z}n)\mathbf{1}(*n((\mathbf{Y}!)(\mathbf{P}n))))\mathbf{1}))\\
&= *\mathbf{3}(*\mathbf{2}((\mathbf{Z}\mathbf{1})\mathbf{1}(*\mathbf{3}((\mathbf{Y}!)(\mathbf{P}\mathbf{1})))))\\
&= *\mathbf{3}(*\mathbf{2}(*\mathbf{1}((\mathbf{Y}!)\mathbf{0})))\\
&= *\mathbf{3}(*\mathbf{2}(*\mathbf{1}((\lambda rn.(\mathbf{Z}n)\mathbf{1}(*n(r(\mathbf{P}n))))(\mathbf{Y}!\mathbf{1}))))\\
&= *\mathbf{3}(*\mathbf{2}(*\mathbf{1}((\lambda n.(\mathbf{Z}n)\mathbf{1}(*n((\mathbf{Y}!)(\mathbf{P}n))))\mathbf{1})))\\
&= *\mathbf{3}(*\mathbf{2}(*\mathbf{1}(\mathbf{Z}\mathbf{0})\mathbf{1}(*\mathbf{0}((\mathbf{Y}!)(\mathbf{P}\mathbf{1})))))\\
&= *\mathbf{3}(*\mathbf{2}(*\mathbf{1}\ \mathbf{1}))\\
&\equiv \mathbf{6}
\end{aligned}
$$

λ in imperative languages

Let's say we had a list of integers, and we want to get a new list that contains each integer in the old list plus 1:

λ in imperative languages

Let's say we had a list of integers, and we want to get a new list that contains each integer in the old list plus 1:

Java:

```
int[] newarray = new int[array.length];
for (int i = 0; i < array.length; i++) {
        newarray[i] = array[i] + 1;
}
```

λ in imperative languages

Let's say we had a list of integers, and we want to get a new list that contains each integer in the old list plus 1:

Java:

```
int[] newarray = new int[array.length];
for (int i = 0; i < array.length; i++) {
        newarray[i] = array[i] + 1;
}
```

Python:

```
newarray = list(map(array, lambda x: x+1))
```

Functional programming (Haskell)


```haskell
reverse' :: [a] -> [a]
reverse' []         = []
reverse' (x:xs) = reverse' xs ++ [x]

elem' :: (Eq a) => a -> [a] -> Bool
elem' _ [] = False
elem' y (x:xs)
    | y == x = True
    | otherwise = elem' y xs

qsort :: (Ord a) => [a] -> [a]
qsort [] = []
qsort (x:xs) = qsort [y | y <- xs, y <= x] ++
               [x] ++ qsort [y | y <- xs, y > x]
```