Lambda Calculus - 2

Greg Michaelson, An Introduction to Functional Programming Through Lambda Calculus, Addison Wesley, 1988.

- **Programming in λ-calculus**
  - **Simulating natural numbers**
- **Recursive functions**
  - **Combinators**
  - **Recursive functions as fixed points**

Programming

Idea: use functions as values to do computation.

Define three useful functions

select_first ≡ $\lambda a.\lambda b.a$

select_second ≡ $\lambda a.\lambda b.b$

cond ≡ $\lambda m.\lambda n.\lambda p.((p\ m)\ n)$, "if p then m else n"

call cond with 3 arguments:

(true_choice)(false_choice)(condition)

Cond

If we apply cond function to select_first and select_second, we obtain:

$$(\lambda p.((p\ e1)\ e2)\ \ \text{select\_first}) = ((\text{select\_first}\ e1)\ e2)$$
$$= e1$$

$$(\lambda p.((p\ e1)\ e2)\ \ \text{select\_second}) =((\text{select\_second}\ e1)\ e2)$$
$$= e2$$

So we can use true ≡ select_first, false ≡ select_second since they pick out the proper arguments for cond to model logical operations.

Natural Numbers

- Can define natural numbers using the idea of a successor function.
  - zero, one is (succ zero), two is (succ one)
- There are many different ways to define zero and succ.
  - zero $\equiv \lambda x.x$
  - succ $\equiv \lambda n.\lambda s.((s\ \text{false})\ n)$
  - one = (succ zero) = $(\ \lambda n.\lambda s.((s\ \text{false})\ n)\ \ \text{zero}\ )$

    $= \lambda s.((s\ \text{false})\ \text{zero})$

Natural Numbers

- Claim (succ zero) is a pair (false, zero) because it acts like this pair when applied to select_first and select_second:

  ((succ zero) select_first)
  = (λs.((s false) zero) select_first)
  = ((select_first false)   zero)
  = (((λa.λb.a) false) zero)
  = false. Similarly,
  ((succ zero) select_second)
  =(λs.((s false) zero) select_second)
  =((select_second false) zero)
  = zero.

Natural Numbers

- Can define numbers two and higher with succ function:

  two = (succ one)
  = (λn.λs.((s false) n) one)
  = λs.((s false) one)
  = λs.{ (s false) (λa.((a false) zero)) }

  three =(succ two)
  = (n. λs.((s false) n) two) =>
  = λs. ((s false) two)
  = λs. ((s false) λa.((a false) one))
  = λs. ((s false) λa. ((a false) λb.((b false) zero)))

Natural Numbers

– Can define the Boolean *iszero* function

- If we apply an arbitrary number to select_first we obtain false, since any number is λs.((s false) prev_num)
- But if we apply zero to select_first we obtain true

  (zero select_first)

  = (λa.a select_first)

  = select_first

  = true
- So *iszero* ≡ λn.(n select_first)

Natural Numbers

- Try pred1 ≡ λn.(n select_second), but then

(pred1 zero) is not well defined because it evaluates to false instead of a number.

- Try pred ≡ λn.(((cond zero) (pred1 n)) (iszero n))

  ≡ if (iszero n) then zero else (pred1 n)
- pred simplifies to λn.(((iszero n) zero) (n select_second)) and this works for zero as well as the other numbers
- We will use the succ and pred functions to build an add function in the natural numbers.

Recursive Functions

- **How about defining the addition of two numbers recursively?**
  - **Define (add x y) by incrementing x and decrementing y until y is zero, so the sum will be accumulated in x.**
  - **(add x y) ≡ ((cond x)(add (succ x)(pred y)))(iszero y)**

Or equivalently,

(add x y) ≡ (if (iszero y) then x else (add (succ x)(pred y)))

This is an explicitly recursive definition that is self-referential!

PROBLEM: How can we stop evaluation?

Defining Add

Use function abstraction to *hide* the recursion. Note that (<fcn> <arg>) is same as λf.(f <arg>) <fcn>

Now we define add1:

add1≡ λf.λx.λy. (if (iszero y) then x else (f (succ x) (pred y)))

Then, add1 is no longer self-referential. But we can't pass add to add1, {add == (add1 add)}, or we get the old definition.

Try add ≡ (add1 add1) and evaluate *(add one two)*.

Defining Add

(add one two) = ((add1 add1) one two)

First let's evaluate (add1 add1)

= (λf.λx.λy. (if (iszero y) then x else (f (succ x) (pred y))) add1

= λx.λy. (if (iszero y) then x else (add1 (succ x) (pred y)))

This doesn't work because we don't have the right arguments necessary for the add1 application. Try again.

add2≡ λf.λx.λy.(if (iszero y)then x else(f f (succ x) (pred y)))

Now evaluate (add2 add2) to see if it works to define add.

(add2 add2) = (λf.λx.λy. (if (iszero y) then x else (f f (succ x) (pred y))) add2

= λx.λy.(if (iszero y)then x else(add2 add2 (succ x) (pred y)))

3 arguments

This looks more promising.

function to apply

Defining Add

false

true

add ≡(add2 add2) in (add one two)

= (λx.λy.(if (iszero y)then x else(add2 add2 (succ x)(pred y))) one two

= if (iszero two) then one else (add2 add2 (succ one) (pred two))

= λf.λx.λy.(if (iszero y)then x else(f f (succ x) (pred y))) add2 (succ one) (pred two)

= if (iszero (pred two))then (succ one) else(add2 add2 (succ (succ one)) (pred (pred two)))

= λf.λx.λy.(if (iszero y)then x else(f f (succ x) (pred y))) add2 (succ (succ one)) (pred (pred two))

= if (iszero(pred (pred two))) then (succ (succ one)) else (add2 add2 (succ (succ (succ one))) (pred (pred (pred two)))

= (succ (succ one)) which is three, as expected!

Functions as Fixed Points

- Seems a heuristic way of simulating recursive functions
- Need a more constructive way to build recursive functions
- *Idea: functions as fixed points*

Let $F(X): 2^N \rightarrow 2^N$ for N , the natural numbers. Define $F(X) = X \cup \{\text{even nos}\}$. Then are there sets $S \subseteq N$ such that $F(S)=S$?

Yes, S={even nos}, S={even nos} ∪ {anything else}

Functions as Fixed Points

– Think of set of Fibonacci number pairs: {(0,1)(1,1)(2,2)(3,3) (4,5) (5,8)…} or factorial pairs {(0,1)(1,1)(2,2)(3,6),…}. What is the smallest set of pairs of numbers from N, that contains these maps?

- ANSWER: the Fibonacci or factorial functions

Combinators

- A *combinator* is a λ-expression with no free variables
- Combinators can be used to define recursive functions as fixed points
- Y combinator, recursive ≡ λf.{ [λs.f(s s)] [λs.f(s s)]} can be used to define our previous *add,* using function abstraction (add1).

Add

(recursive_add1) ≡ (λf.[λs.f (s s)] [λs.f (s s)]) add1

≡ [λs.add1 (s s)] [λs.add1 (s s)]

≡ add1([λs.add1 (s s)] [λs.add1 (s s)])

≡ {λf.λx.λy. (if (iszero y) then x else (f (succ x)(pred y)))}
([λs.add1 (s s)] [λs.add1 (s s)])

≡ {λx.λy. (if (iszero y) then x else
(([λs.add1 (s s)] [λs.add1 (s s)]) (succ x)(pred y)))}

Now can try ({recursive_add1} one two) to see what is calculated. Compare this calculation with that on the previous slide #12.

Add, again

false

true

({recursive_add1} one two)
= ({λx.λy. (if (iszero y) then x else (([λs.add1 (s s)] [λs.add1 (s s)])
(succ x)(pred y)))} one two)
= (if (iszero two) then one else (([λs.add1 (s s)] [λs.add1 (s s)])
(succ one)(pred two)))
…= add1([λs.add1 (s s)] [λs.add1 (s s)]) (succ one) (pred two)
= (if (iszero (pred two)) then (succ one) else (([λs.add1 (s s)]
[λs.add1 (s s)]) (succ (succ one)) (pred (pred two))))
= add1([λs.add1 (s s)] [λs.add1 (s s)]) (succ (succ one)) (pred (pred two))
= (if (iszero (pred (pred two))) then (succ (succ one)) else (([λs.add1 (s s)]
[λs.add1 (s s)]) (succ (succ(succ one))) (pred (pred(pred two)))))
= (succ (succ one)) *or* ***three***

Fixed Points

- Fixed point of a function, f(a)=a.
- Examples of fixed points

f	fixed point
λx.6	6
λx.6-x	3
$\lambda x.x^{2}+x-4$	2,-2
λx.x	every value
λx.x+1	no value

Fixed Points

- Suppose ∃ function Y such that YF = f for F an arbitrary function and f its fixed point.

  $F=\lambda g.\lambda n.$ if $n<3$ then 1 else $g(n-1)+g(n-2)$

  then (F fib) is the recursive definition of the Fibonacci sequence

  $\lambda n.$ if $n<3$ then 1 else $fib(n-1)+ fib(n-2)$

  but F itself has no explicit mention of fib.

Y Combinator

- Claim, for arbitrary function g, $z = (\lambda x.g(x\ x))(\lambda x.g(x\ x))$, has the property that $g(z)=z$.

$$\begin{aligned} z &= (\lambda x.g(x\ x))(\lambda x.g(x\ x)) \\ &= g\{(\lambda x.g(x\ x))(\lambda x.g(x\ x))\} \\ &= g(z). \end{aligned}$$

- Y combinator: $Y= \lambda h.((\lambda x.h(x\ x))\ (\lambda x.h(x\ x)))$
- If Y combinator evaluation results in a normal form, Church Rosser theorem ensures *uniqueness* of the function so defined.

Y Combinator

- **Does it work?**

$Y\ (\lambda x.6) = \lambda h.((\lambda x.h(x\ x))\ (\lambda x.h(x\ x)))\ \lambda x.6$

$= (\lambda x.(\lambda x.6)(x\ x))\ (\lambda x.(\lambda x.6)(x\ x))$

$= (\lambda x.6)\ ((\lambda x.(\lambda x.6)(x\ x))\ (\lambda x.(\lambda x.6)(x\ x)))$

$= 6$ evaluated by name; similarly

$Y\ (\lambda x.x+1) = (\lambda x.(\lambda x.x+1)(x\ x))\ (\lambda x.(\lambda x.x+1)(x\ x))$

$= (\lambda x.x+1)\ ((\lambda x.(\lambda x.x+1)(x\ x))\ (\lambda x.(\lambda x.x+1)(x\ x)))$

$= (\lambda x.x+1)\ \{Y\ (\lambda x.x+1)\}$

$= \{Y\ (\lambda x.x+1)\} + 1$

This is an f such that Yf = Yf+1. There is no such function whose return value is an integer, so a fixed point does not exist here. Use bottom symbol ⊥ to indicate this.

Y Combinator

- **Can evaluate the fixed points which are functions by giving them arguments**
  - **Y F 3 will return the 3rd Fibonacci number**

$Y\ F\ 3 = (\lambda x.F(x\ x))\ (\lambda x.F(x\ x))\ 3$, (let term in red be K)

$= (\lambda x.\lambda g.\lambda n.((\text{if } n<3 \text{ then } 1 \text{ else } g(n-1)+g(n-2))\ (x\ x)))\ K\ 3$

$= (\lambda g.\lambda n.(\text{if } n<3 \text{ then } 1 \text{ else } g(n-1)+g(n-2))\ (K\ \ K)))\ \ 3$

$= (\lambda n.\text{if } n<3 \text{ then } 1 \text{ else } (K\ K\ )(n-1)+(K\ \ K)(n-2))\ \ 3$

$= \text{if } 3<3 \text{ then } 1 \text{ else } (K\ K\ )(3-1)+(K\ \ K)(3-2)$

$= \text{if } 3<3 \text{ then } 1 \text{ else } (K\ K)\ (2) + (K\ K)\ (1)$, cont.

Y Combinator

(K K) 2 = ((λx.F(x x)) K) 2
= (λx.λg.λn.((if n<3 then 1 else g(n-1)+g(n-2)) (x x))) K 2
= (λg.λn.(if n<3 then 1 else g(n-1)+g(n-2)) (K K))) 2
= (λn.if n<3 then 1 else (K K)(n-1)+(K K)(n-2)) 2
= 1

(K K) 1 = ((λx.F(x x)) K) 1
= (λx.λg.λn.((if n<3 then 1 else g(n-1)+g(n-2)) (x x))) K 1
= (λg.λn.(if n<3 then 1 else g(n-1)+g(n-2)) (K K))) 1
= (λn.if n<3 then 1 else (K K)(n-1)+(K K)(n-2)) 1
= 1

Therefore, Y F 3 is 1+1= 2!