Foundations and Interpreters for Functional Programming

Lambda Calculus and Lisp

Motivation

We have seen some of the core concepts of functional programming:

- programs as functions from input to output, no need for mutation
- program evaluation as substitution
- defining computation using (recursive) functions
- recursive data types such as lists and trees
- higher-order functions, which take other functions as arguments

Next: how to **implement** a **minimalistic** functional language ourselves

Among the simplest functional languages: **Lisp** (and its variant, **Scheme**)

Mathematical foundation of languages like Lisp: **Lambda Calculus**

Review: Creating and Applying Functions in Scala

Creating a function:

```
val  f =   (x:Any) => x
```

Applying a function:

```
f(5)                    // gives 5
```

Creating and applying in the same expression (without giving name to f):

```
((x:Any) => x)(5)       // gives 5
```

anonymous function

Creating a function is an operation, just like e.g. adding two numbers

Curried Functions in Scala through Examples

Creating a function that returns another function:

```
val  g = (x:Any) => ((y:Any) => x)        // given x, return constant function x
val  h = (x:Any) => ((y:Any) => y)        // given x, return identity function
```

Applying a function

```
g(5)               // gives some function reference
(g(5))(7)          // gives 5: first make “constant 5” function, then apply it
g(5)(7)            // gives 5, means exactly the same as above
h(5)(7)            // gives 7: ignore 5: make identity function, then apply it

((x:Any) => ((y:Any) => x)) (5) (7) =     // by substitution of x with 5
 (((y:Any) => 5)) (7)  =                  // by substitution of y with 7
 5                                        // no y in body, so 7 disappeared
```

A Minimal Language

What if the only two constructs we had were:

- applying a function
- creating a function

What could we compute in such a programming language?

Lambda Calculus

Church, A., **1932**, "A set of postulates for the foundation of logic", *Annals of Mathematics* (2nd Series), 33(2): 346–366.

Scala equivalent	**Lambda calculus**
((x : Any) => ((y:Any) => x)) (a)(b)	((λx. (λy. x)) a) b

Lambda calculus has only variables (x,y,a,b,…) and these two constructs:

	Scala	**Lambda calculus**
1. **application**	f(x)	f x
2. **lambda abstraction** (=function creation)	(x:Any) => M	λx. M

Example: Evaluation in Scala vs Lambda Calculus

Scala notation:

```
((x:Any) => ((y:Any) => x)) (a) (b) =      // by substitution of x with a
            ((y:Any) => a) (b)  =          // by substitution of y with b
                        a                  // no y in body, so b disappeared
```

Lambda calculus notation:

```
((λ x. (λ y. x)) a) b     =                // by substitution of x with a
      (λ y. a) b  =                        // by substitution of y with b
          a                                // no y in body, so b disappeared
```

One Rule for Evaluation

$(\lambda x.M)N$ = "term obtained from M by replacing every x with N"

This rule is called **beta reduction** (β-reduction)

Examples:

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

$(\lambda x.b)N = b$

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

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

More Notation and Examples

To indicate that terms are equal because of beta reduction, we use $\Rightarrow_\beta$

$(\lambda x.M)N \Rightarrow_\beta$ "term obtained from M by replacing all x occurrences with N"

If $M \Rightarrow_\beta M'$ we assume equality $M=M'$ also holds

Functions have one argument, we use currying. We use these abbreviations:

λ x y. M N	≡	λx.(λy.(MN))
f M N	≡	((f M) N)

Examples:

(≡ means same term up to our shorthands)

$(\lambda x.\ x)\ (a\ b) \Rightarrow_\beta a\ b$

$(\lambda x\ y.\ c\ x)\ a\ b \equiv ((\lambda x.\ (\lambda y.\ (c\ x)))\ a)\ b \Rightarrow_\beta (\lambda y.\ (c\ a))\ b \Rightarrow_\beta c\ a$

$(\lambda\ f\ x.\ f(f\ x))\ (\lambda y.\ a)\ b \Rightarrow_\beta (\lambda y.\ a)((\lambda y.\ a)\ b) \Rightarrow_\beta a$

Consequence of Our Notation

$(\lambda\, x_1\, x_2 \ldots\, x_k\, .\, M)\; N_1\, N_2 \ldots N_k \quad \Rightarrow_\beta^{+} \quad M'$

($\Rightarrow^{+}_\beta$ means some number of $\Rightarrow_\beta$ steps)

where M’ is the term obtained from M by replacing

x_1 with N_1 then

x_2 with N_2 then

… and, finally,

x_k with N_k.

A Minimal Language

Only two operations

- applying a function: $f\ x$
- creating a function: $\lambda x.\ M$

What can we compute in such a programming language?

λ Calculus Can Represent: Booleans

We represent Boolean value b as the function corresponding to "if (b)"

if (true) M N should evaluate to M

if (false) M N should evaluate to N

So, **define**

true ≡ $\lambda x\ y.\ x$ so: true M N = $(\lambda x\ y.\ x)$ M N = M

false ≡ $\lambda x\ y.\ y$ so: false M N = $(\lambda x\ y.\ y)$ M N = N

So instead of **if (b) M N** we just write **b M N**

b will take M and N, returning M or N, depending on if is true or false

Example: Implementing Disjunction

Write function that implements logical "or" on such booleans

Solution: we would like to define function or such that

or p q = **if** (p) true **else** q

Given our implementation of **if** as application of p, the definition is:

or ≡ λp q. p true q

Example:

or false true = (λp q. p true q) false true
= false true true
≡ (λx y. y) true true
= true

λ Calculus Can Represent: Pairs

Pair is something from which we can get the first and the second element

Define

$(M,N) \equiv \lambda f.\ f\ M\ N$

$P.\_1 \equiv P\ (\lambda x\ y.\ x)$ to extract first element, apply pair to $(\lambda x\ y.\ x)$

$P.\_2 \equiv P\ (\lambda x\ y.\ y)$

Why does this work?

$(M,N).\_1 \equiv (\lambda f.\ f\ M\ N)\ (\lambda x\ y.\ x) = (\lambda x\ y.\ x) M\ N = M$

$(M,N).\_2 \equiv (\lambda f.\ f\ M\ N)\ (\lambda x\ y.\ y) = (\lambda x\ y.\ y) M\ N = N$

λ Calculus Can Represent: Lists

A list is something we can match on and deconstruct if it is not empty:

list **match** {
  case Nil => **M**
  case Cons(x,y) => **N_0**   // N_0 refers to x and y
}

will be represented as application: (list M (λ x y. N_0))

We define list as a function that will take such **M** and **N** ≡ λ x y. N_0 as arguments

Nil ≡ λm n. m   so: Nil M N ≡ (λm n. m) M N = M

Cons P Q ≡ λm n. n P Q   (we could have defined: cons ≡ λ p q m n. n p q)

so: (Cons P Q) M N ≡ (λm n. n P Q) M N = N P Q = (λ x y. N_0) P Q

Compute Head of List

```
(a :: b) match {
  case Nil => Z
  case Cons(x,y) => x
}
```

is represented as:

$(\lambda m\ n.\ n\ a\ b)\ Z\ (\lambda x\ y.\ x)$

and evaluates, as expected to:

$(\lambda x\ y.\ x)\ a\ b$ and then to: a

Return pair (tail,tail) if list non-empty, else Z

```
list match {
  case Nil => Z
  case Cons(x,y) => (y,y)
}
```

is represented using our lambda calculus encoding by:

list Z (λ x y. (λf. f y y))

Computation that takes any number of steps

$(\lambda x.\ x\ x)\ (\lambda x.\ x\ x) \Rightarrow_\beta (\lambda x.\ x\ x)\ (\lambda x.\ x\ x) \Rightarrow_\beta \ldots$ loops.

More usefully: $(\lambda x.\ F\ (x\ x))\ (\lambda x.\ F\ (x\ x)) \Rightarrow_\beta F\ ((\lambda x.\ F\ (x\ x))(\lambda x.\ F\ (x\ x)))$

If we denote $Y_F = (\lambda x.\ F\ (x\ x))\ (\lambda x.\ F\ (x\ x))$ **(for each term F)**

Then $Y_F \Rightarrow_\beta F\ ((\lambda x.\ F\ (x\ x))\ (\lambda x.\ F\ (x\ x))) = F\ Y_F$ i.e. $Y_F \Rightarrow_\beta F(Y_F)$

A recursive function uses itself in its body (typically applies it):

def h(x:Any) = P(h(Q(x)),x) for some P and Q

def h = ((x:Any) => P(h(Q(x)),x))

Denote right-hand side of the last **def** by F(h), since x is a bound variable

def h = F(h) to unfold recursion, replace h by F(h) in body

We define $h = Y_F$ so $h = Y_F \Rightarrow_\beta F\ Y_F \Rightarrow_\beta F(F\ Y_F) = F(F\ h) \Rightarrow_\beta \ldots$

Replace all list elements by Z: List(1,2,3) $\rightarrow$ List(Z,Z,Z)

```
def mkZ(list) = list match {
  case Nil => Nil
  case Cons(x,y) => Cons(Z, mkZ(y))
}
```

After encoding match, still using recursion

$$mkZ = \lambda list.\ list\ Nil\ (\lambda\ x\ y.\ Cons\ Z\ (mkZ\ y))$$

After encoding recursion, it becomes mkZ = Y_F

for $F \equiv \lambda self.\ \lambda list.\ list\ Nil\ (\lambda\ x\ y.\ Cons\ Z\ (self\ y))$

So mkZ can be defined as Y_F which, in this case, is:

$(\lambda x.\ (\lambda\ self.\ \lambda list.\ list\ Nil\ (\lambda\ x\ y.\ Cons\ Z\ (self\ y)))\ \ (x\ x))$
$(\lambda x.\ (\lambda\ self.\ \lambda list.\ list\ Nil\ (\lambda\ x\ y.\ Cons\ Z\ (self\ y)))\ \ (x\ x))$

Example execution of mkZ on argument

Given $\quad F \equiv \lambda self.\ \lambda list.\ list\ Nil\ (\lambda\ x\ y.\ Cons\ Z\ (self\ y))$

$mkZ\ (Cons\ a\ Nil) \equiv Y_F\ (Cons\ a\ Nil) = F\ Y_F\ (Cons\ a\ Nil)$

$\equiv (\lambda \mathbf{self}.\ \lambda list.\ list\ Nil\ (\lambda\ x\ y.\ Cons\ Z\ (\mathbf{self}\ y)))\ \mathbf{Y_F}\ (Cons\ a\ Nil)$

replacing $\quad \mathbf{self} \rightarrow \mathbf{Y_F} \qquad list \rightarrow Cons\ a\ Nil$

$= (Cons\ a\ Nil)\ Nil\ (\lambda\ x\ y.\ Cons\ Z\ (\mathbf{Y_F}\ y))$

$\equiv (\lambda m\ n.\ n\ a\ Nil)\ Nil\ (\lambda\ x\ y.\ Cons\ Z\ (\mathbf{Y_F}\ y))$

$= (\lambda\ x\ y.\ Cons\ Z\ (\mathbf{Y_F}\ y))\ a\ Nil = Cons\ Z\ (\mathbf{Y_F}\ Nil)$

$= Cons\ Z\ ((F\ Y_F)\ Nil) = Cons\ Z\ (((\lambda self.\ \lambda \mathbf{list}.\ \mathbf{list}\ Nil\ (\lambda\ x\ y.\ Cons\ Z\ (self\ y)))\ Y_F)\ \mathbf{Nil})$

$= Cons\ Z\ (\mathbf{Nil}\ Nil\ (\lambda\ x\ y.\ Cons\ Z\ (Y_F\ y))) = Cons\ Z\ Nil$

because $\quad Nil\ m\ n = m$