Programming Abstractions

Week 3-2: Combinators and combinatory logic

Stephen Checkoway

An early 20th century crisis in mathematics

Russell's Paradox

Define *S* to be the set of all sets that are *not* elements of themselves

- $S = \{x \mid x \notin x\}$

Is *S* an element of *S*?

- Assume so: $S \in S \implies S \notin S$ by the definition of *S*, a contradiction
- Assume not: $S \notin S \implies S \in S$ by the definition of *S*, another contradiction!

This led to a hunt for a non-set-theoretic foundation for mathematics

- Combinatory logic (Moses Schönfinkel and rediscovered by Haskell Curry)
- Lambda calculous (Alonzo Church and others)
  - This forms the basis for functional programming!

Combinatory term

One of three things

A variable (from an infinite list of possible variables)

- I'll use lowercase, upright letters: e.g., f, g, h, x, y, z

A combinator (a function that operates on functions)

- One of the three primitive functions
  - Identity: (I x) = x
  - Constant: (K x y) = x
  - Substitution: (S f g x) = (f x (g x))
- A new combinator C = E where E is a combinatory term, e.g.,
  - J = (S K K)
  - B = (S (K S) K)

(E_1 E_2) An application of a combinatory term E_1 to term E_2

- Application is left-associative so (E_1 E_2 E_3 E_4) is (((E_1 E_2) E_3) E_4)

The primitive combinators

The identity combinator (I x) = x

- Given any combinatory term x, it returns x

The constant combinator (K x y) = x

- I.e., ((K x) y) = x which you can think of as (K x) returns a function that given any argument y returns x

The substitution combinator (S f g x) = (f x (g x))

- You can think of S as taking two functions f and g and some term x. f is applied to x which returns a function and that function is applied to the result of (g x)
- But really, f, g, and x are all just combinatory terms

What is the result of applying the constant combinator in the combinatory term (K z I)

- (I x) = x
- (K x y) = x
- (S f g x) = (f x (g x))

A. The variable z

B. The combinator I

C. The combinatory term (z I)

D. It's an error because I takes an argument but none is provided

E. None of the above

What is the result of applying the substitution combinator in the combinatory term (S (f x) h y z)

- (I x) = x
- (K x y) = x
- (S f g x) = (f x (g x))

A. The variable f

B. The combinator S

C. The combinatory term ((f x) y (h y) z)

D. The combinatory term (f x (h x) y z)

E. It's an error because S takes 3 arguments but is given four

Expressing S, K, and I in Racket

```
(define (I x)
  x)

(define (K x)
  (λ (y) x))

(define (S f)
  (λ (g)
    (λ (x)
      ((f x) (g x)))))
```

Using the combinators (in Racket)

```
((K 25) 37) ; returns 25

; ((curry-* x) y) is just (* x y)
(define (curry-* x)
  (λ (y)
    (* x y)))

(define (square x)
  (((S curry-*) I) x))
```

As combinators we get (S * I x) = (* x (I x)) = (* x x)

Equivalence between Scheme and combinatory logic

We can represent combinators in Scheme as procedures with no free variables (i.e., every variable used in the body of the procedure is a parameter)

There are no λs in combinatory logic so no way to make new functions

However, combinatory logic does have a way to get the same effect as λ expressions

- We won't cover this, but we can convert every expression in λ calculus into combinatory logic
- λ calculus is Turing-complete (it can perform any computation) so combinatory logic is as well!

Example of a new combinator

L = (S K)

- (I x) = x
- (K x y) = x
- (S f g x) = (f x (g x))

Example of a new combinator

L = (S K)

Apply the rules to the left-most combinator in each step, starting with (L x y)

- (I x) = x
- (K x y) = x
- (S f g x) = (f x (g x))

Example of a new combinator

L = (S K)

Apply the rules to the left-most combinator in each step, starting with (L x y)

- (I x) = x
- (K x y) = x
- (S f g x) = (f x (g x))

(L x y) = ((S K) x y) [Definition of L]

Example of a new combinator

L = (S K)

- (I x) = x
- (K x y) = x
- (S f g x) = (f x (g x))

Apply the rules to the left-most combinator in each step, starting with (L x y)

(L x y)	= ((S K) x y)	[Definition of L]
	= (S K x y)	[Constant]

Example of a new combinator

L = (S K)

Apply the rules to the left-most combinator in each step, starting with (L x y)

- (I x) = x
- (K x y) = x
- (S f g x) = (f x (g x))

(L x y) = ((S K) x y) [Definition of L]
= (S K x y) [Constant]
= (K y (x y)) [Substitution]

Example of a new combinator

L = (S K)

Apply the rules to the left-most combinator in each step, starting with (L x y)

- (I x) = x
- (K x y) = x
- (S f g x) = (f x (g x))

(L x y)	= ((S K) x y)	[Definition of L]
	= (S K x y)	[Constant]
	= (K y (x y))	[Substitution]
	= y	[Constant]

Example: Diagonalizing combinator

W = (S S L)

- (I x) = x
- (K x y) = x
- (S f g x) = (f x (g x))
- (L x y) = y

Example: Diagonalizing combinator

W = (S S L)

Apply the rules to the left-most combinator in each step, starting with (W f x)

- (I x) = x
- (K x y) = x
- (S f g x) = (f x (g x))
- (L x y) = y

Example: Diagonalizing combinator

W = (S S L)

Apply the rules to the left-most combinator in each step, starting with (W f x)

(W f x) = ((S S L) f x) [Definition of W]

- (I x) = x
- (K x y) = x
- (S f g x) = (f x (g x))
- (L x y) = y

Example: Diagonalizing combinator

W = (S S L)

Apply the rules to the left-most combinator in each step, starting with (W f x)

- (I x) = x
- (K x y) = x
- (S f g x) = (f x (g x))
- (L x y) = y

(W f x)	= ((S S L) f x)	[Definition of W]
	= (S S L f x)	[Associativity]

Example: Diagonalizing combinator

W = (S S L)

Apply the rules to the left-most combinator in each step, starting with (W f x)

- (I x) = x
- (K x y) = x
- (S f g x) = (f x (g x))
- (L x y) = y

(W f x)	= ((S S L) f x)	[Definition of W]
	= (S S L f x)	[Associativity]
	= (S f (L f) x)	[Substitution]

Example: Diagonalizing combinator

W = (S S L)

Apply the rules to the left-most combinator in each step, starting with (W f x)

- (I x) = x
- (K x y) = x
- (S f g x) = (f x (g x))
- (L x y) = y

(W f x)	= ((S S L) f x)	[Definition of W]
	= (S S L f x)	[Associativity]
	= (S f (L f) x)	[Substitution]
	= (f x ((L f) x))	[Substitution]

Example: Diagonalizing combinator

W = (S S L)

Apply the rules to the left-most combinator in each step, starting with (W f x)

- (I x) = x
- (K x y) = x
- (S f g x) = (f x (g x))
- (L x y) = y

(W f x)	= ((S S L) f x)	[Definition of W]
	= (S S L f x)	[Associativity]
	= (S f (L f) x)	[Substitution]
	= (f x ((L f) x))	[Substitution]
	= (f x (L f x))	[Associativity]

Example: Diagonalizing combinator

W = (S S L)

Apply the rules to the left-most combinator in each step, starting with (W f x)

(W f x) = ((S S L) f x)	[Definition of W]
= (S S L f x)	[Associativity]
= (S f (L f) x)	[Substitution]
= (f x ((L f) x))	[Substitution]
= (f x (L f x))	[Associativity]
= (f x x)	[Applying L]

- (I x) = x
- (K x y) = x
- (S f g x) = (f x (g x))
- (L x y) = y

Example: Composition combinator

B = (S (K S) K)

(B f g x)	= ((S (K S) K) f g x)	[Definition of B]
	= (S (K S) K f g x)	[Associativity]
	= ((K S) f (K f) g x)	[Substitution]
	= (K S f (K f) g x)	[Associativity]
	= (S (K f) g x)	[Constant]
	= ((K f) x (g x))	[Substitution]
	= (K f x (g x))	[Associativity]
	= (f (g x))	[Constant]

- (I x) = x
- (K x y) = x
- (S f g x) = (f x (g x))

Work out what J = (S K K) does in (J x)

Apply the rules of the left most combinator in each step, starting with (J x)

- (I x) = x
- (K x y) = x
- (S f g x) = (f x (g x))

I is unnecessary

Since (S K K x) is always x, (S K K) and I are *functionally equivalent*

- (I x) = x
- (K x y) = x
- (S f g x) = (f x (g x))

We can replace I in any combinatory term with (S K K)

Since we can model all computation using S, K, and I and I can be built from S and K, S and K are sufficient for any computation!

Unlambda is a programming language built out of S, K, function application, and functions for printing and reading a character

- Hello world! in Unlambda: `````````````.H.e.l.l.o.,. .w.o.r.l.d.!i
- Echo user input: ```sii```si`k`ci`@|

The Y-combinator

How do we write a recursive function?

Easy, use `define`

```
(define len
  (λ (lst)
    (cond [(empty? lst) 0]
          [else (add1 (len (rest lst)))])))
```

For the rest of this lecture, we're not going to use `(define (fun args) …)`

How do we write a recursive function?

(without using define)

Easy, use `letrec`

```
(letrec ([len
          (λ (lst)
            (cond [(empty? lst) 0]
                  [else (add1 (len (rest lst)))]))])
  len)
```

Recall, this binds `len` to our function `(λ (lst) …)` in the body of the `letrec`

This expression returns the procedure bound to `len` which computes the length of its argument

How do we write a recursive function?

(just using anonymous functions created via λs)

Less easy, but let's give it a go!

```
(λ (lst)
  (cond [(empty? lst) 0]
        [else (add1 (??? (rest lst)))]))
```

We need to put something in the recursive case in place of the `???` but what?

If we replace the `???` with
`(λ (lst) (error "List too long!"))`
we'll get a function that correctly computes the length of empty lists, but fails with nonempty lists

Put the function itself there?

```
(λ (lst)
  (cond [(empty? lst) 0]
        [else (add1 ((λ (lst)
                       (cond [(empty? lst) 0]
                             [else (add1 (??? (rest lst)))]))
                     (rest lst)))]))
```

Not a terrible attempt, we still have ???, but now we can compute lengths of the empty list and a single element list.

Maybe we can abstract out the function

```
(λ (len)
  (λ (lst)
    (cond [(empty? lst) 0]
          [else (add1 (len (rest lst)))])))
```

This isn't a function that operates on lists!

It's a function that takes a function `len` as a parameter and returns a closure that takes a list `lst` as a parameter and computes a sort of length function using the passed in `len` function

make-length

```
(define make-length
  (λ (len)
    (λ (lst)
      (cond [(empty? lst) 0]
            [else (add1 (len (rest lst)))]))))
```

This is the same function as before but bound to the identifier `make-length`

- The orange text is the body of `make-length`
- The purple text is the body of the closure returned by `(make-length len)`

```
(define L0 (make-length (λ (lst) (error "too long"))))
```

- `L0` correctly computes the length of the empty list but fails on longer lists

make-length

```
(define make-length
  (λ (len)
    (λ (lst)
      (cond [(empty? lst) 0]
            [else (add1 (len (rest lst)))]))))

(define L0 (make-length (λ (lst) (error "too long"))))
(define L1 (make-length L0))
(define L2 (make-length L1))
(define L3 (make-length L2))
```

- `Ln` correctly computes the length of lists of size at most n
- We need an `L∞` in order to work for all lists
- `(make-length length)` would work correctly, but that's cheating!

Enter the Y combinator

Y is a "fixed-point combinator"

- Y = (S (K (S I I)) (S (S (K S) K) (K (S I I))))

If `f` is a function of one argument, then `(Y f) = (f (Y f))`

```
(Y make-length)
=> (make-length (Y make-length))
=> (λ (lst)
     (cond [(empty? lst) 0]
           [else (add1 ((Y make-length) (rest lst)))]))
```

This is precisely the length function: `(define length (Y make-length))`

How is this length?

How is this length?

Let's step through applying our length function to '(1 2 3)

How is this length?

Let's step through applying our length function to '(1 2 3)

```
(length '(1 2 3)) ; so lst is bound to '(1 2 3)
```

How is this length?

Let's step through applying our length function to '(1 2 3)

```
(length '(1 2 3)) ; so lst is bound to '(1 2 3)
=> (cond [(empty? lst) 0]
         [else (add1 ((Y make-length) (rest lst)))])
```

How is this length?

Let's step through applying our length function to '(1 2 3)

```
(length '(1 2 3)) ; so lst is bound to '(1 2 3)
=> (cond [(empty? lst) 0]
         [else (add1 ((Y make-length) (rest lst)))])
=> (add1 (length '(2 3))) ; lst is bound to '(2 3)
```

How is this length?

Let's step through applying our length function to '(1 2 3)

```
(length '(1 2 3)) ; so lst is bound to '(1 2 3)
=> (cond [(empty? lst) 0]
         [else (add1 ((Y make-length) (rest lst)))])
=> (add1 (length '(2 3))) ; lst is bound to '(2 3)
=> (add1 (cond [(empty? lst) 0]
               [else (add1 ((Y make-length) (rest lst)))]))
```

How is this length?

Let's step through applying our length function to '(1 2 3)

```
(length '(1 2 3)) ; so lst is bound to '(1 2 3)
=> (cond [(empty? lst) 0]
         [else (add1 ((Y make-length) (rest lst)))])
=> (add1 (length '(2 3))) ; lst is bound to '(2 3)
=> (add1 (cond [(empty? lst) 0]
               [else (add1 ((Y make-length) (rest lst)))]))
=> (add1 (add1 (length '(3)))) ; lst is bound to '(3)
```

How is this length?

Let's step through applying our length function to '(1 2 3)

```
(length '(1 2 3)) ; so lst is bound to '(1 2 3)
=> (cond [(empty? lst) 0]
         [else (add1 ((Y make-length) (rest lst)))])
=> (add1 (length '(2 3))) ; lst is bound to '(2 3)
=> (add1 (cond [(empty? lst) 0]
               [else (add1 ((Y make-length) (rest lst)))]))
=> (add1 (add1 (length '(3)))) ; lst is bound to '(3)
=> (add1 (add1 (cond […][else (add1 …)])))
```

How is this length?

Let's step through applying our length function to '(1 2 3)

```
(length '(1 2 3)) ; so lst is bound to '(1 2 3)
=> (cond [(empty? lst) 0]
         [else (add1 ((Y make-length) (rest lst)))])
=> (add1 (length '(2 3))) ; lst is bound to '(2 3)
=> (add1 (cond [(empty? lst) 0]
               [else (add1 ((Y make-length) (rest lst)))]))
=> (add1 (add1 (length '(3)))) ; lst is bound to '(3)
=> (add1 (add1 (cond […][else (add1 …)])))
=> (add1 (add1 (add1 (length '())))) ; lst is bound to '()
```

How is this length?

Let's step through applying our length function to '(1 2 3)

```
(length '(1 2 3)) ; so lst is bound to '(1 2 3)
=> (cond [(empty? lst) 0]
         [else (add1 ((Y make-length) (rest lst)))])
=> (add1 (length '(2 3))) ; lst is bound to '(2 3)
=> (add1 (cond [(empty? lst) 0]
               [else (add1 ((Y make-length) (rest lst)))]))
=> (add1 (add1 (length '(3)))) ; lst is bound to '(3)
=> (add1 (add1 (cond […][else (add1 …)])))
=> (add1 (add1 (add1 (length '())))) ; lst is bound to '()
=> (add1 (add1 (add1 (cond [(empty? lst) 0][…]))))
```

How is this length?

Let's step through applying our length function to '(1 2 3)

```
(length '(1 2 3)) ; so lst is bound to '(1 2 3)
=> (cond [(empty? lst) 0]
         [else (add1 ((Y make-length) (rest lst)))])
=> (add1 (length '(2 3))) ; lst is bound to '(2 3)
=> (add1 (cond [(empty? lst) 0]
               [else (add1 ((Y make-length) (rest lst)))]))
=> (add1 (add1 (length '(3)))) ; lst is bound to '(3)
=> (add1 (add1 (cond […][else (add1 …)])))
=> (add1 (add1 (add1 (length '())))) ; lst is bound to '()
=> (add1 (add1 (add1 (cond [(empty? lst) 0][…]))))
=> (add1 (add1 (add1 0)))
```

How is this length?

Let's step through applying our length function to '(1 2 3)

```
(length '(1 2 3)) ; so lst is bound to '(1 2 3)
=> (cond [(empty? lst) 0]
         [else (add1 ((Y make-length) (rest lst)))])
=> (add1 (length '(2 3))) ; lst is bound to '(2 3)
=> (add1 (cond [(empty? lst) 0]
               [else (add1 ((Y make-length) (rest lst)))]))
=> (add1 (add1 (length '(3)))) ; lst is bound to '(3)
=> (add1 (add1 (cond […][else (add1 …)])))
=> (add1 (add1 (add1 (length '())))) ; lst is bound to '()
=> (add1 (add1 (add1 (cond [(empty? lst) 0][…]))))
=> (add1 (add1 (add1 0)))
=> 3
```

How is this length?

Let's step through applying our length function to '(1 2 3)

```
(length '(1 2 3)) ; so lst is bound to '(1 2 3)
=> (cond [(empty? lst) 0]
         [else (add1 ((Y make-length) (rest lst)))])
=> (add1 (length '(2 3))) ; lst is bound to '(2 3)
=> (add1 (cond [(empty? lst) 0]
               [else (add1 ((Y make-length) (rest lst)))]))
=> (add1 (add1 (length '(3)))) ; lst is bound to '(3)
=> (add1 (add1 (cond […][else (add1 …)])))
=> (add1 (add1 (add1 (length '())))) ; lst is bound to '()
=> (add1 (add1 (add1 (cond [(empty? lst) 0][…]))))
=> (add1 (add1 (add1 0)))
=> 3
```

But wait, how can that work?

Two problems:

- We defined Y in terms of Y! It's recursive and the whole point was to write recursive anonymous functions
  - Not quite, Y = (S (K (S I I)) (S (S (K S) K) (K (S I I)))), but we still need to write this in Racket
- `(Y f) = (f (Y f))` but then
  `(f (Y f)) = (f (f (Y f)) = (f (f (f (Y f)))) = …`
  and this will never end

Defining Y

```
(define Y
  (λ (f)
    ((λ (g) (f (g g)))
     (λ (g) (f (g g))))))
```

It's tricky to see what's going on but `Y` is a function of `f` and its body is applying the anonymous function `(λ (g) (f (g g)))` to the argument `(λ (g) (f (g g)))` and returning the result.

```
(Y foo) = ((λ (g) (foo (g g)))              ; By applying Y to foo
           (λ (g) (foo (g g))))
        = (foo ((λ (g) (foo (g g)))         ; By applying orange fun
                (λ (g) (foo (g g)))))       ;   to purple argument
        = (foo (Y foo))                     ; From definition of Y
```

Never ending computation

This form of the Y-combinator doesn't work in Scheme because the computation would never end

We can fix this by using the related Z-combinator

```
(define Z
  (λ (f)
    ((λ (g) (f (λ (v) ((g g) v))))
     (λ (g) (f (λ (v) ((g g) v)))))))
```

This is the argument to our recursive function

With this definition, we can create a length function

```
(define length (Z make-length))
```

We can use Z to make recursive functions

Given a recursive function of one variable

```
(define foo
  (λ (x) … (foo …) …))
```

we can construct this only using anonymous functions by way of Z

```
(Z (λ (foo) (λ (x) … (foo …) …)))
```

```
Factorial
(Z (λ (fact)
      (λ (n)
        (if (zero? n)
            1
            (* n (fact (sub1 n)))))))
```

What about multi-argument functions?

We can use apply!

This is the list of arguments to our recursive function

```
(define Z*
  (λ (f)
    ((λ (g) (f (λ args (apply (g g) args))))
     (λ (g) (f (λ args (apply (g g) args)))))))
```