Programming Abstractions

Week 4-1: 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

Equivalence between Racket and combinatory logic

We can show that Racket and SKI combinatory logic are equally powerful models of computation

We need to show two things

- Show that we can represent SKI combinators in Racket
- Show that we can represent Racket in combinatory logic

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)

Expressing Racket in Combinatory logic

Needs a few steps

Church–Turing thesis: Any function on the natural numbers can be computed if and only if it can be computed by a Turing machine (take CSCI 383!)

In the 1930s, Church, Turing, and Kleene proved these are all equivalent

- General recursive functions
- Turing machines
- Lambda calculus

Lambda calculus forms the basis for LISP (and thus Scheme and Racket)

We can convert Racket → lambda calculus → SKI

Racket → Combinatory logic (example)

Consider the Racket expression `(λ (x) (λ (f) (f x)))`
This is a curried function of two arguments where the second, `f`, is a function that gets applied to the first, `x`

In lambda calculus, this would be written $\lambda x.\lambda f.(f\ x)$

Through a mechanical transformation (check out the Wikipedia article on combinatory logic for details), this becomes (S (K (S I)) (S (K K) I))

You can prove this example is correct by applying the definitions of SKI to (S (K (S I)) (S (K K) I) x f) and you'll end up with (f x)

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)

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

- (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]
= (S K x y) [Associativity]

Example of a new combinator

L = (S K)

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) [Associativity]
= (K y (x y)) [Substitution]

- (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]
= (S K x y)	[Associativity]
= (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)

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

- (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]
= (S S L f x) [Associativity]
= (S f (L f) x) [Substitution]

- (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]
= (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))

- (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) until no more rules apply

A. (x x)

B. (K x)

C. (K x K x)

D. x

E. (x K)

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`@|
  ```