Lambda Calculus and Computation

6.037 - Structure and Interpretation of Computer Programs

Benjamin Barenblat

bbaren@mit.edu Massachusetts Institute of Technology With material from Mike Phillips, Nelson Elhage, and Chelsea Voss

January 30, 2019

Benjamin Barenblat
Lambda Calculus and Computation

Equivalence of Computation Methods

First part of the proof: Church-Turing thesis.

Any intuitive notion for a "computer" that you can come up with will be no more powerful than a Turing machine or than lambda calculus. That is, most models of computation are equivalent.

Turing-complete means capable of simulating Turing machines.

Lambda calculus is Turing-complete (proof: later), and Turing machines can simulate lambda calculus.

Some others:

- *Turing machines* are Turing-complete
- *Scheme* is Turing-complete
- Minecraft is Turing-complete
- Conway's Game of Life is Turing-complete
- Wolfram's Rule 110 cellular automaton is Turing-complete

Limits to Computation

David Hilbert's *Entscheidungsproblem* (1928): Build a calculating machine that gives a yes/no answer to all mathematical questions.



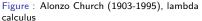




Figure : Alan Turing (1912-1954), Turing machines

Theorem (Church, Turing, 1936): These models of computation can't solve every problem. Proof: next!

Benjamin Barenblat

Lambda Calculus and Computation

6.037

6.037

Does not compute?

- Are there problems which our notion of computing cannot solve?
- Reworded: are there *functions* that cannot be computed?
- Consider functions which map naturals to naturals.
- Can write out a function f as the infinite list of naturals f(0), f(1), f(2)...
- Any program text can be written as a single number, joining together this list

Does not compute?

How many uncomputable problems?

- Now consider every possible function
- Put them in a big table, one function per row, one input per column
- Diagonalize!
- We get a contradiction: here's a function that's not in your list.

Theorem (Church, Turing): These models of computation can't solve every problem.

- Countably infinite: ℵ₀
 - The number of naturals
 - The number of binary strings
 - The number of programs
- Uncountably infinite: 2^{\aleph_0}
 - The number of functions mapping from natural to natural

Benjamin Barenblat 6.037
Lambda Calculus and Computation

Benjamin Barenblat
Lambda Calculus and Computation

Does not compute: Halting Problem

Aside: what does this do?

Okay, but can you give me an example?

- We've seen our programs create infinite lists and infinite loops
- Can we write a program to check if an expression will return a value?

```
(define (halt? p)
; ...
)
```

```
((lambda (x) (x x))
(lambda (x) (x x)))
= ((lambda (x) (x x))
(lambda (x) (x x)))
= ((lambda (x) (x x))
(lambda (x) (x x)))
= ...
```

Lambda Calculus and Computation

7

Benjamin Barenblat 6.037

Does not compute: Halting Problem

The Source of Power

Contradiction!

```
(define (troll)
  (if (halt? troll)
    ; if halts? says we halt, infinite-loop
       ((lambda (x) (x x)) (lambda (x) (x x)))
    ; if halts? says we don't, return a value
    #f))
(halt? troll)
```

Halting Problem is undecidable for Turing Machines – and thus all programming languages. (Turing, 1936)

Want to learn more computability theory? See 18.400 J/6.045 J or 18.404 J/6.840 J (Sipser).

What's the minimal set of Scheme syntax that you need to achieve Turing-completeness?

- define
- set!
- numbers
- strings
- if
- recursion
- cons
- booleans
- lambda

Benjamin Barenblat

Lambda Calculus and Computation

0.037

Lambda Calculus and Computation

6.037

Cons cells?

```
(define (cons a b)
  (lambda (c)
        (c a b)))
(define (car p)
        (p (lambda (a b) a)))
(define (cdr p)
        (p (lambda (a b) b)))
```

Booleans?

Benjamin Barenblat

```
(define true
  (lambda (a b)
      (a)))

(define false
  (lambda (a b)
      (b)))

(define if
  (lambda (test then else)
      (test then else))
Also try: and, or, not
```

Numbers?

Numbers?

Number N: A procedure which takes in a successor function s and a zero z, and returns the successor applied to the zero N times.

- For example, 3 is represented as (s(s(sz))), given s and z
- This technique: *Church numerals*

```
(define church-0
 (lambda (s z)
   z))
(define (church-1
 (lambda (s z)
    (s z)))
(define (church-2
 (lambda (s z)
   (s (s z))))
```

Benjamin Barenblat

Lambda Calculus and Computation

Benjamin Barenblat Lambda Calculus and Computation

Numbers?

Let, define?

```
(define (church-inc n)
 (lambda (s z)
    (s (n s z))))
(define (church-add a b)
 (lambda (s z)
    (a s (b s z))))
(define (also-church-add a b)
 (a church-inc b))
```

For fun: Write decrement, write multiply.

```
(define x 4)
(...stuff)
becomes...
((lambda (x)
  (...stuff)
) 4)
```

Lambda Calculus and Computation

Use lambdas.

Let, define?

Factorial again

```
A problem arises!

(define (fact n)
    (if (= n 0)
        1
        (* n (fact (- n 1)))))

Why? (lambda (fact) ...) (...definition of fact...) fails! fact is not yet defined when called in its function body.

If we can't name "fact" how do we use it in the recursive call?
```

Benjamin Barenblat Lambda Calculus and Computation 6.037

Lambda Calculus and Computation

6.03

Now without define

Messy. Can we do better?

Producing Fixed Points

Now let's define Y as: (lambda (f) ((lambda (g) (f (g g))) (lambda (g) (f (g g))))) We'll show that (Y f) = (f (Y f)) – that we can use Y to create fixed points.

Benjamin Barenblat

<u>Lambda Calculus and C</u>omputation

6.037

Lambda Calculus and Computation

6.037

Producing Fixed Points

Now we can define fact as follows:

Can create fact without using define!
Can create all of Scheme using just lambda!
Lambda calculus is Turing-complete! Church—Turing thesis!

Producing Fixed Points

From the problem before: we want a fixed point of generate-fact.

Fun links

- https://xkcd.com/505/
- http://www.lel.ed.ac.uk/~gpullum/loopsnoop.html
- https://youtu.be/1X21HQphy6I
- https://youtu.be/My8AsV7bA94
- https://youtu.be/xP5-iIeKXE8
- https://en.wikipedia.org/wiki/Rule_110