Computation

Chelsea Voss <csvoss@mit.edu>
6.037
IAP 2017

With thanks to Mike Phillips and Nelson Elhage

Church-Turing Thesis

Church-Turing Thesis

- If a function can be computed by an algorithm, then it must also be computable by a Turing Machine
- And vice-versa
- Thus Java, Scheme, Python, etc, are all equivalent in the functions they can compute

Does not compute?

- Are there functions that cannot be computed?
- Consider functions which map integers to integers.
- Can write out a function $f$ as the infinite list of integers $f(0), f(1), f(2), ...$
- Any program text can be written as a single number
  - Read off the bytes as a big number

Some fun Turing machines

- Minecraft Turing Machine
  - https://youtu.be/1X21HQphy6I
- Game of Life Turing Machine
  - https://youtu.be/My8AsV7bA94
- Game of Life in Game of Life
  - https://youtu.be/xP5-ileKXE8
- Rule 110, a Turing-complete cellular automaton

Does not compute?

- We can write a table of ALL PROGRAMS

<table>
<thead>
<tr>
<th>Prog</th>
<th>Function it computes</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>$f_0(0), f_0(1), f_0(2), f_0(3), ...$</td>
</tr>
<tr>
<td>1</td>
<td>$f_1(0), f_1(1), f_1(2), f_1(3), ...$</td>
</tr>
<tr>
<td>2</td>
<td>$f_2(0), f_2(1), f_2(2), f_2(3), ...$</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
Does not compute?

<table>
<thead>
<tr>
<th>Prog</th>
<th>Function it computes</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>$f_0(0), f_0(1), f_0(2), f_0(3), \ldots$</td>
</tr>
<tr>
<td>1</td>
<td>$f_1(0), f_1(1), f_1(2), f_1(3), \ldots$</td>
</tr>
<tr>
<td>2</td>
<td>$f_2(0), f_2(1), f_2(2), f_2(3), \ldots$</td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
</tbody>
</table>

Construct a new function, $g$:

\[
g(0) = f_0(0) + 1 \\
g(1) = f_1(1) + 1 \\
\ldots
\]

Where is it in our table?

How many uncomputable?

- **Countably infinite**: $\kappa_0$
  - The number of integers
  - The number of binary strings
  - The number of programs
- **Uncountably infinite**: $2^{(\kappa_0)}$
  - The number of maps from integer to integer
  - The number of sets of binary strings
  - The number of problem specifications

Halting Problem

- Undecidable for Turing machines!
  - Alan Turing (1936)

- A clever poem-proof on the subject:
  - [http://www.lel.ed.ac.uk/~gpullum/loopsnoop.html](http://www.lel.ed.ac.uk/~gpullum/loopsnoop.html)

- Want to learn more computability theory?
  - 18.404J/6.840J (Sipser)

Does not compute

- A more interesting function we cannot compute?
- $f(x) = 1$ if $x$ represents a program that halts, 0 otherwise

\[
\text{(define (halt? p)} \\
\text{...)}
\]

The Source of Power

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

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

Booleans?

```
(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)))
```

```
(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?

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

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

(define if
  (lambda (test then else)
    ((test then) else)))

Numbers?

- Number N: A procedure which takes in a successor function and a zero and returns the successor applied to the zero N times
  - For example, 3 is represented as (s (s (s z))), given s and z
  - This is called Church numerals

(define if (lambda (test then else)
            ((test then) else))

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

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

(define if
  (lambda (test then else)
    ((test then) else)))

(numbers)

● Number N: A procedure which takes in a successor function and a zero and returns the successor applied to the zero N times
  - For example, 3 is represented as (s (s (s z))), given s and z
  - This is called Church numerals

(define (church-0
  (lambda (s)
    (lambda (z)
      z)))

(define (church-1
  (lambda (s)
    (lambda (z)
      (s z))))

(define (church-2
  (lambda (s)
    (lambda (z)
      (s (s z))))))

(define (church-inc n)
  (lambda (s)
    (lambda (z)
      ((n s) z))))

(define (church-add a b)
  ((a church-inc) b))

(define (also-church-add a b)
  ((a church-inc) b)))

For fun: Write decrement, write multiply.
Use lambdas.

(define x 4)
(...stuff)

→

((lambda (x)
  (...stuff)
) 4)

PROBLEM!

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

PROBLEM!

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

(lambda (fact) …) (...definition of fact…)
fails!

If we can't name “fact” how do we use it in the recursive call?
Aside: What does this do?

\[ (\text{lambda} (x) \ (x \ x)) \\
(\text{lambda} (x) \ (x \ x)) \]

Messy. Can we do better?

Let's define \texttt{fact-inner} as:

\[ (\text{lambda} (\text{fact}) \\
(\text{lambda} (n) \\
(\text{if} \ (= \ n \ 0) \ \\
1 \\
(* \ n \ (\text{fact} \ (- \ n \ 1)))))) \]

Huh--
What's \texttt{(fact-inner fact)}?

Factorial again

Run it with a copy of itself.

\texttt{(define (fact inner-fact n) \\
(if (= n 0) \ \\
1 \\
(* n \\
(inner-fact inner-fact (- n 1))))} \\
(fact fact 4) works!

Messy. Can we do better?

Let's define \texttt{fact-inner} as:

\texttt{(lambda (fact) \\
(lambda (n) \\
(if (= n 0) \\
1 \\
(* n (fact fact (- n 1))))))}

Huh--
What's \texttt{(fact-inner fact)}?

\texttt{(fact-inner fact) = fact}

Now without define

\texttt{(fact 4) becomes:}

\begin{verbatim}
( (lambda (fact n) \\
(if (= n 0) \\
1 \\
(* n (fact fact (- n 1))))))

(lambda (fact n) \\
(if (= n 0) \\
1 \\
(* n (fact fact (- n 1)))))

(fact fact 4)
\end{verbatim}

Messy. Can we do better?

Let's define \texttt{fact-inner} as:

\begin{verbatim}
(\text{lambda} (\text{fact}) \\
(\text{lambda} (n) \\
(\text{if} \ (= \ n \ 0) \ \\
1 \\
(* \ n \ (\text{fact} \ (- \ n \ 1)))))) \\
\end{verbatim}

Huh--
What's \texttt{(fact-inner fact)}?

\texttt{(fact-inner fact) = fact}

A fixed point
Producing Fixed Points

Now let's define \( Y \) as:

\[
Y := (\lambda (f) \left( \lambda (g) (f (g g)) \right) \right) \left( \lambda (g) (f (g g)) \right) \)

What does \( Y \) do?

Let \( H := (\lambda (g) (\text{fact-inner} \ (g g))) \).

Let \( \text{fact} = (Y \ \text{fact-inner)} \).

Is \( \text{fact} = (\text{fact-inner} \ \text{fact})? \)

\[
\text{fact} \Rightarrow (Y \ \text{fact-inner})
\]

What does \( Y \) do?

Let \( H := (\lambda (g) (\text{fact-inner} \ (g g))) \).

Let \( \text{fact} = (Y \ \text{fact-inner}) \).

Is \( \text{fact} = (\text{fact-inner} \ \text{fact})? \)

\[
\text{fact} \Rightarrow (Y \ \text{fact-inner})
\]

What does \( Y \) do?

Let \( H := (\lambda (g) (\text{fact-inner} \ (g g))) \).

Let \( \text{fact} = (Y \ \text{fact-inner}) \).

Is \( \text{fact} = (\text{fact-inner} \ \text{fact})? \)

\[
\text{fact} \Rightarrow (Y \ \text{fact-inner})
\]

What does \( Y \) do?

Let \( H := (\lambda (g) (\text{fact-inner} \ (g g))) \).

Let \( \text{fact} = (Y \ \text{fact-inner}) \).

Is \( \text{fact} = (\text{fact-inner} \ \text{fact})? \)

\[
\text{fact} \Rightarrow (Y \ \text{fact-inner})
\]

\[
\Rightarrow (H \ H)
\]

What does \( Y \) do?

Let \( H := (\lambda (g) (\text{fact-inner} \ (g g))) \).

Let \( \text{fact} = (Y \ \text{fact-inner}) \).

Is \( \text{fact} = (\text{fact-inner} \ \text{fact})? \)

\[
\text{fact} \Rightarrow (Y \ \text{fact-inner})
\]

\[
\Rightarrow (H \ H)
\]

\[
\Rightarrow (\lambda (g) (\text{fact-inner} \ (g g))) \ H)
\]
What does Y do?

\[ Y := (\lambda (f) \ ( (\lambda (g) \ (f \ (g \ g)) \)) \ (\lambda (g) \ (f \ (g \ g)))) \] 

Let \( H := (\lambda (g) \ (\text{fact-inner} \ (g \ g))) \).

Let \text{fact} = (Y \ \text{fact-inner}).

Is \text{fact} = (\text{fact-inner} \ \text{fact})?

\[
\begin{align*}
\text{fact} \\
\rightarrow (Y \ \text{fact-inner}) \\
\rightarrow (H \ H) \\
\rightarrow (\lambda (g) \ (\text{fact-inner} \ (g \ g))) \ H \\
\rightarrow (\text{fact-inner} \ (H \ H))
\end{align*}
\]

Equivalent to: \( (\text{fact-inner} \ (Y \ \text{fact-inner})) \)

The Y Combinator

\[ Y := (\lambda (f) \ ( (\lambda (g) \ (f \ (g \ g)) \)) \ (\lambda (g) \ (f \ (g \ g)))) \] 

\[ (Y \ F) = (F \ (Y \ F)) \]

Lambda calculus is Turing-complete! 

\textit{Church-Turing thesis!}

What does Y do?

\[ Y := (\lambda (f) \ ( (\lambda (g) \ (f \ (g \ g)) \)) \ (\lambda (g) \ (f \ (g \ g)))) \] 

Let \( H := (\lambda (g) \ (\text{fact-inner} \ (g \ g))) \).

Let \text{fact} = (Y \ \text{fact-inner}).

Is \text{fact} = (\text{fact-inner} \ \text{fact})?

\[
\begin{align*}
\text{fact} \\
\rightarrow (Y \ \text{fact-inner}) \\
\rightarrow (H \ H) \\
\rightarrow (\lambda (g) \ (\text{fact-inner} \ (g \ g))) \ H \\
\rightarrow (\text{fact-inner} \ (H \ H))
\end{align*}
\]

Equivalent to: \( (\text{fact-inner} \ (Y \ \text{fact-inner})) \)

Equivalent to: \( (\text{fact-inner} \ \text{fact}) \)