Limits to Computation

David Hilbert’s *Entscheidungsproblem* (1928)
Limits to Computation

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

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

Figure: Alonzo Church (1903-1995), lambda calculus

Figure: Alan Turing (1912-1954), Turing machines
Limits to Computation

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

**Figure:** Alonzo Church (1903-1995), lambda calculus

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

**Theorem** (Church, Turing, 1936): These models of computation can’t solve every problem.
Limits to Computation

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

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

First part of the proof: Church–Turing thesis.
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.
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.
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.
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
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
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
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
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
Are there problems which our notion of computing cannot solve?
Does not compute?

- Are there problems which our notion of computing cannot solve?
- Reworded: are there functions that cannot be computed?
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.
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 $f(0), f(1), f(2)\ldots$
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?

- Now consider every possible function
Does not compute?

- Now consider every possible function
- Put them in a big table, one function per row, one input per column
Now consider every possible function
Put them in a big table, one function per row, one input per column
Diagonalize!
Does not compute?

- 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.
Does not compute?

- 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.*
How many uncomputable problems?

- Countably infinite: $\aleph_0$
How many uncomputable problems?

- Countably infinite: $\aleph_0$
  - The number of naturals
How many uncomputable problems?

- Countably infinite: \( \aleph_0 \)
  - The number of naturals
  - The number of binary strings
How many uncomputable problems?

- Countably infinite: $\aleph_0$
  - The number of naturals
  - The number of binary strings
  - The number of programs
How many uncomputable problems?

- Countably infinite: \( \aleph_0 \)
  - The number of naturals
  - The number of binary strings
  - The number of programs
- Uncountably infinite: \( 2^{\aleph_0} \)
How many uncomputable problems?

- Countably infinite: $\aleph_0$
  - 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
Okay, *but can you give me an example?*
Okay, but can you give me an example?

- We’ve seen our programs create infinite lists and infinite loops
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?
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)
  ; ...
)
Aside: what does this do?

\[
\begin{align*}
((\text{lambda} \ (x) \ (x \ x)) \\
(\text{lambda} \ (x) \ (x \ x)))
\end{align*}
\]
Aside: what does this do?

\[
\begin{align*}
&\ (\text{lambda} \ (x) \ (x \ x)) \\
&\ (\text{lambda} \ (x) \ (x \ x))
\end{align*}
\]

\[
= \ (\text{lambda} \ (x) \ (x \ x)) \\
(\text{lambda} \ (x) \ (x \ x))
\]
Aside: what does this do?

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

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

= \[
((\text{lambda } (x) \ (x \ x)) \\
(\text{lambda } (x) \ (x \ x)))
\]
Aside: what does this do?

\[
\begin{align*}
((\lambda (x) (x \ x)) \\
(\lambda (x) (x \ x))) \\
= ((\lambda (x) (x \ x)) \\
(\lambda (x) (x \ x))) \\
= ((\lambda (x) (x \ x)) \\
(\lambda (x) (x \ x))) \\
= \ldots
\end{align*}
\]
Does not compute: Halting Problem

Contradiction!
Does not compute: Halting Problem

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))
Does not compute: Halting Problem

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)
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)
Contradiction!

```scheme
(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 18.400J/6.045J or 18.404J/6.840J (Sipser).
The Source of Power

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

- define
- set!
- numbers
- strings
- if
- recursion
- cons
- booleans
- lambda
Cons cells?

(define (cons a b)
  (lambda (c)
    (c a b)))
(define (cons a b)
  (lambda (c)
    (c a b)))

(define (car p)
  (p (lambda (a b) a)))
(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 b)
        (a)))

Also try:
and, or, not
Booleans?

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

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

Also try: and, or, not
Booleans?

(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
Booleans?

(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?

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 $(s (s (s z)))$, given $s$ and $z$
- This technique: *Church numerals*
Numbers?

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

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

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

(define (church-2
  (lambda (s z)
    (s (s z)))))
(define (church-inc n)
  (lambda (s z)
    (s (n s z)))))
(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 (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))
Numbers?

(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.
Let, define?

Use lambdas.
Let, define?

Use lambdas.

(define x 4)
(...stuff)
Use lambdas.

\[(\text{define } x \ 4)\]
\[(\ldots \text{stuff})\]

becomes...

\[((\lambda (x)\]
\[\quad (\ldots \text{stuff})\]
\[\quad )\ 4)\]
A problem arises!

(define (fact n)
  (if (= n 0)
    1
    (* n (fact (- n 1))))
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.
A problem arises!

\[
\text{(define (fact n)}
\begin{align*}
&\quad \text{(if (= n 0)} \\
&\qquad 1 \\
&\qquad (* n (fact (- n 1))))
\end{align*}
\]

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?
Factorial again

Run it with a copy of itself.

\[
\text{(define (fact inner-fact n)}
\text{    (if (= n 0)}
\text{        1}
\text{        (* n}
\text{            (* n}
\text{                (inner-fact inner-fact (- n 1)))))}
\]
Factorial again

Run it with a copy of itself.

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

Now, (fact fact 4) works!
Now without define

\[(\text{fact fact 4}) \text{ becomes:} \]
Now without define

(fact fact 4) becomes:

(((lambda (inner-fact n)
    (if (= n 0)
      1
      (* n (inner-fact inner-fact (- n 1)))))
  (lambda (inner-fact n)
    (if (= n 0)
      1
      (* n (inner-fact inner-fact (- n 1)))))
  4)
Messy. Can we do better?

Let’s define generate-fact as:
Let's define generate-fact as:

\[
\text{(lambda (inner-fact)
  (lambda (n)
    (if (= n 0)
      1
      (* n (inner-fact (- n 1)))))))
\]
Let’s define generate-fact as:

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

Huh – what’s \( \text{(generate-fact fact)} \)?
Let’s define generate-fact as:

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

Huh – what’s (generate-fact fact)?
\[(\text{generate-fact fact}) = \text{fact.}\]
Messy. Can we do better?

Let’s define `generate-fact` as:

```
(lambda (inner-fact)
  (lambda (n)
    (if (= n 0)
      1
      (* n (inner-fact (- n 1)))))))
```

Huh – what’s `(generate-fact fact)`?

`(generate-fact fact) = fact.

A fixed point!"
Now let's define $Y$ as:

$$(\lambda (f) ((\lambda (g) (f (g \ g))) (\lambda (g) (f (g \ g)))))$$

We'll show that $(Y \ f) = (f \ (Y \ f))$
Now let's define $Y$ as:

$$(\lambda f \quad ((\lambda g \quad (f (g \ g))) \quad ((\lambda g \quad (f (g \ g))) \quad ((\lambda g \quad (f (g \ g)))))))$$

We'll show that $(Y \ f) = (f \ (Y \ f))$ – that we can use $Y$ to create fixed points.
Producing Fixed Points

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

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

```
(define Y (lambda (f)
    ((lambda (g) (f (g g)))
     (lambda (g) (f (g g))))))
```

;; For convenience:
;; H := (lambda (g) (f (g g)))

;; Is (generate-fact (Y generate-fact))
;; = (Y generate-fact)?
;; (Y generate-fact)
;; = (H H) ; (with f = generate-fact)
;; = (generate-fact (H H))
;; = (generate-fact (Y generate-fact)) ; Success!
Producing Fixed Points

Now we can define fact as follows:

```
(Y (lambda (inner-fact)
  (lambda (n)
    (if (= n 0) 1 (* n (inner-fact (- n 1)))))))
```
Producing Fixed Points

Now we can define fact as follows:

\[(Y \ (\lambda (inner\text{-}\text{fact}) \ \\
\quad (\lambda (n) \ \\
\quad \quad (if \ (= \ n \ 0) \ \\
\quad \quad \quad 1 \ \\
\quad \quad \quad (* \ n \ (inner\text{-}\text{fact} \ (- \ n \ 1))))))))\]
Producing Fixed Points

Now we can define \texttt{fact} as follows:

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

Can create \texttt{fact} without using \texttt{define}!
Producing Fixed Points

Now we can define fact as follows:

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

Can create fact without using define!
Can create all of Scheme using just lambda!
Now we can define fact as follows:

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

Can create fact without using define!
Can create all of Scheme using just lambda!

**Lambda calculus is Turing-complete!**
Now we can define \texttt{fact} as follows:

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

Can create \texttt{fact} without using \texttt{define}!
Can create all of Scheme using just \texttt{lambda}!
\textbf{Lambda calculus is Turing-complete!} Church–Turing thesis!
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