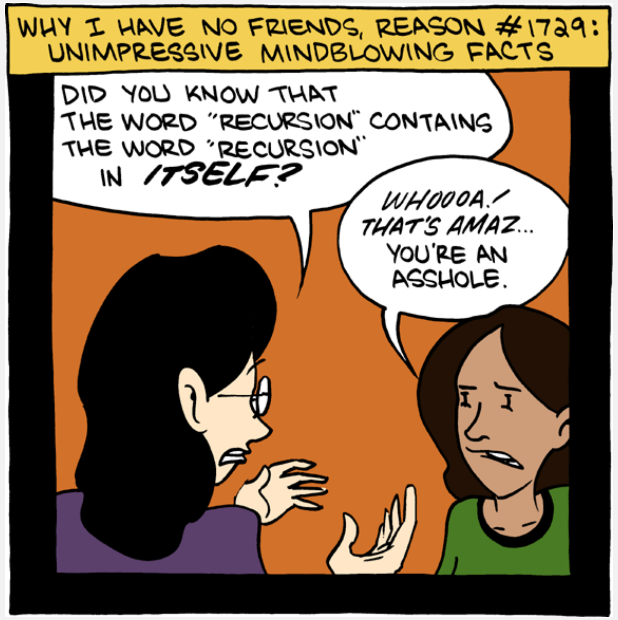
UMB CS622

Turing Machines and Recursion

Monday, November 1, 2021



Announcements

- Hw6 extended deadline:
  - due Wed 11/3 11:59pm

Recursion in Programming

```
(define (factorial n)
  (if (zero? n)
      1
      (* n (factorial (sub1 n)))))
```

In most programming languages, you can call a function recursively, <u>even before it's completely defined</u>!

Turing Machines and Recursion

- We've been saying: "**A Turing machine models programs.**"

- **<u>Q</u>**: Is a recursive program modeled by a Turing machine?

- **<u>A</u>**: Yes!
  - But it's not explicit.
  - In fact, it's a little complicated.
  - Need to prove it ...

- **<u>Today</u>**: The Recursion Theorem

A ***Turing machine*** is a 7-tuple, $(Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject})$, where Q, Σ, Γ are all finite sets and

1. Q is the set of states,
2. Σ is the input alphabet not containing the ***blank symbol*** $\sqcup$,
3. Γ is the tape alphabet, where $\sqcup \in \Gamma$ and $\Sigma \subseteq \Gamma$,
4. $\delta : Q \times \Gamma \longrightarrow Q \times \Gamma \times \{L, R\}$ is the transition function,
5. $q_0 \in Q$ is the start state,
6. $q_{accept} \in Q$ is the accept state, and
7. $q_{reject} \in Q$ is the reject state, where $q_{reject} \neq q_{accept}$.

Where's the recursion in this definition???

The Recursion Theorem

- You can write a TM description like this:

B = "On input w:

1. Obtain, via the recursion theorem, own description $\langle B \rangle$.

The Recursion Theorem

Example Use Case

$A_{\mathsf{TM}} = \{\langle M, w\rangle \mid M \text{ is a TM and } M \text{ accepts } w\}$

Prove A_{TM} is undecidable, by contradiction:

assume that Turing machine H decides A_{TM}

B = "On input w:

1. Obtain, via the recursion theorem, own description $\langle B\rangle$.
2. Run H on input $\langle B, w\rangle$.
3. Do the opposite of what H says. That is, *accept* if H rejects and *reject* if H accepts."

	$\langle M_1\rangle$	$\langle M_2\rangle$	$\langle M_3\rangle$	$\langle M_4\rangle$	$\cdots$	$\langle D\rangle$
M_1	*accept*	*reject*	*accept*	*reject*		*accept*
M_2	*accept*	*accept*	*accept*	*accept*	$\cdots$	*accept*
M_3	*reject*	*reject*	*reject*	*reject*		*reject*
M_4	*accept*	*accept*	*reject*	*reject*		*accept*
$\vdots$			$\vdots$		$\ddots$	
D	*reject*	*reject*	*accept*	*accept*		?

This is the non-existent "D" machine the TM that **does the opposite of itself, defined using recursion!**
(prev. defined using diagonalization)

How can a TM “obtain it’s own description?”

How does a TM even know about “itself” before it’s completely defined?

A Simpler Exercise

Idea:
TMs can receive TMs as input;
Just assume input will be yourself!

Our Task:

- Create a TM that, without using recursion, prints itself.
  - How does this TM get knowledge about "itself"?

- An example, in English:

"TM input"

"TM"

Print out two copies of the following, the second one in quotes:
"Print out two copies of the following, the second one in quotes:"

"argument"
(the TM gets itself from its input!)

- This TM knows about "itself",
  - but it does not explicitly use recursion!

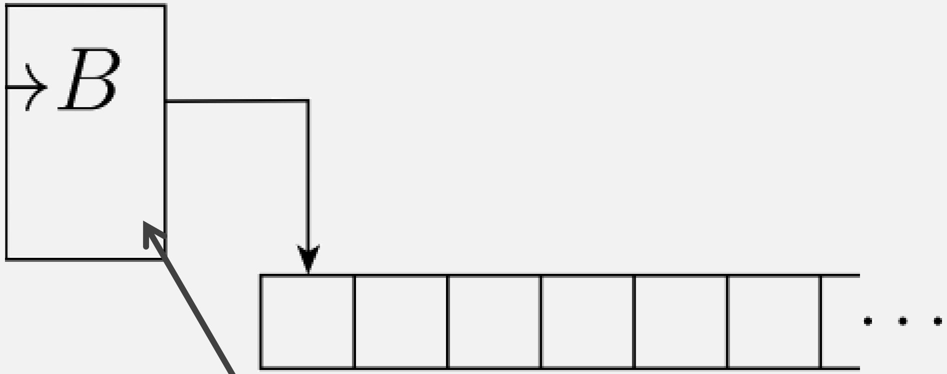
Self-Printing Turing Machine

q creates a TM (that prints a string) [1], and outputs it as a string (i.e., it's "quoted") [2]

So q(<M>) prints a "quoted" M

The following TM Q computes $q(w)$.

Q = "On input string w:
1. Construct the following Turing machine P_w.
   P_w = "On any input: [1]
   1. Erase input.
   2. Write w on the tape.
   3. Halt."
2. Output $\langle P_w \rangle$." [2]



"argument" (the TM itself, encoded as string)

"TM"

"TM input" (will be itself)

B = "On input $\langle M \rangle$, where M is a portion of a TM:
1. Compute $q(\langle M \rangle)$. — Second (quoted) copy
2. Combine the result with $\langle M \rangle$ to make a complete TM. — First copy
3. Print the description of this TM and halt."

Print out two copies of the following, the second on in quotes:

SELF, Defined With The Recursion Theorem

$SELF$ = "On any input:
  1. Obtain, via the recursion theorem, own description $\langle SELF \rangle$.
  2. Print $\langle SELF \rangle$."

- So a TM doesn't need explicit recursion to call itself!
- What about TMs that do more than "print itself"?

The Recursion Theorem, Formally

Recursion theorem Let T be a Turing machine that computes a function $t\colon \Sigma^* \times \Sigma^* \longrightarrow \Sigma^*$. There is a Turing machine R that computes a function $r\colon \Sigma^* \longrightarrow \Sigma^*$, where for every w,

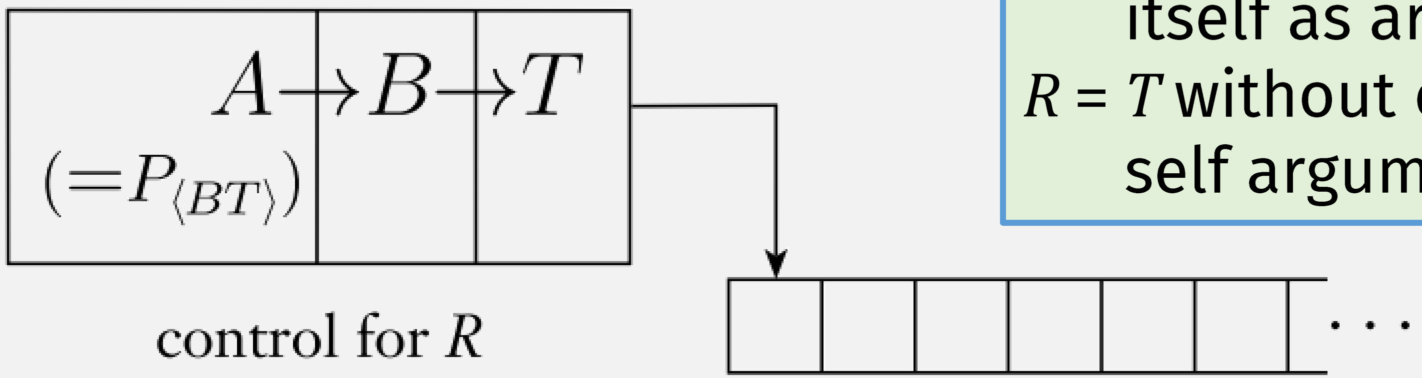
$$r(w) = t(\langle R \rangle, w).$$

<u>In English</u>:

- If you want a TM R that can "obtain own description" ...
- ... instead create a TM T with an extra "itself" argument ...
- ... then construct R from T **???**

The Recursion Theorem, Pictorially

- To convert a "*T*" to "*R*":



AB = *SELF* (prev slide)
T = machine that gets itself as argument
R = *T* without explicit self argument

1. Construct *A* = program constructing <*BT*>, and
2. Pass result to *B* (from before),
3. which passes "itself" to *T*

Recursion Theorem, A Concrete Example

- If you want:
  - Recursive fn

```
(define (factorial n) ;; R
  (if (zero? n)
      1
      (* n (factorial (sub1 n)))))
```

- Instead create:
  - Non-recursive fn

```
(define (factorial/itself ITSELF n) ;; T
  (if (zero? n)
      1
      (* n (ITSELF (sub1 n)))))
```

Recursion Theorem says you can convert

It's not clear how the recursion theorem applies to **real programs**?

TMs and Recursive Programs

- So a TM doesn't need explicit recursion to call itself!
- **What about programs? (TMs = Programs)**
- Can we write **recursive programs** **<u>without</u> using explicit recursion?**

Interlude: Lambda

A (very high-level) Turing Machine

- λ = anonymous function, e.g. `(λ (x) x)`
  - **C++**: `[](int x){ return x; }`
  - **Java**: `(x) -> { return x; }`
  - **Python**: `lambda x : x`
  - **JS**: `(x) => { return x; }`

A Self-Printing Program

Print out two copies of the following, the second one in quotes:
“Print out two copies of the following, the second one in quotes:”

“function”

“parameter”

“argument”

```
((λ (SELF) (print2x SELF))
 "(λ (SELF) (print2x SELF))")
```

Could we write a program that does **something other than print “itself”**?

```
(define (print2x str)
  (printf "(~a\n ~v)\n" str str))
```

(could have inlined this)

First copy

Second copy (quoted)

Non-Printing Uses of *SELF*

- Program that prints "itself":

```
((λ (SELF) (print2x SELF))
 "(λ (SELF) (print2x SELF))")
```

eta-expansion:
Any function f = λx.(f x)

- Program that runs "itself" repeatedly (i.e., it infinite loops):

```
((λ (SELF) (SELF SELF))
 (λ (SELF) (SELF SELF)))
```

Call arg fn with itself as arg

Don't convert arg to string

- Loop, but do something useful each time?

"package up" the recursion

```
((λ (SELF) (f (SELF SELF)))
 (λ (SELF) (f (SELF SELF))))
```

```
(λ (f)
  ((λ (SELF) (f (λ (v) ((SELF SELF) v))))
   (λ (SELF) (f (λ (v) ((SELF SELF) v))))))
```

- **None of these programs use explicit recursion!**

Y combinator

Recursion Theorem Proof: Coding Demo

- Program that passes "itself" to another function:

Y combinator

```
(λ (f)
  ((λ (x) (f (λ (v) ((x x) v))))
   (λ (x) (f (λ (v) ((x x) v))))))
```

Pass to

- Function that needs "itself"

```
(define (factorial/itself ITSELF n) ;; T
  (if (zero? n)
      1
      (* n (ITSELF (sub1 n)))))
```

Fixed Points

- A value x is a fixed point of a function f if $f(x) = x$

Recursion Theorem and Fixed Points

Let $t: \Sigma^* \longrightarrow \Sigma^*$ be a computable function. Then there is a Turing machine F for which $t(\langle F \rangle)$ describes a Turing machine equivalent to F. Here we'll assume that if a string isn't a proper Turing machine encoding, it describes a Turing machine that always rejects immediately.

In this theorem, t plays the role of the transformation, and F is the fixed point.

PROOF Let F be the following Turing machine.

F = "On input w:
1. Obtain, via the recursion theorem, own description $\langle F \rangle$.
2. Compute $t(\langle F \rangle)$ to obtain the description of a TM G.
3. Simulate G on w."

Clearly, $\langle F \rangle$ and $t(\langle F \rangle) = \langle G \rangle$ describe equivalent Turing machines because F simulates G.

Fixed point is a **TM that is unchanged** by the function

- I.e., Recursion Theorem implies:
  - "every TM that computes on TMs has a fixed point"
  - As code: "every function on functions has a fixed point"

Y Combinator

- `mk-recursive-fn` = a "fixed point finder"

```
(define mk-recursive-fn
  (λ (f)
    ((λ (x) (f (λ (v) ((x x) v))))
     (λ (x) (f (λ (v) ((x x) v)))))))
```

- **factorial** is the fixed point of **mk-factorial**

Summary: Where "Recursion" Comes From

- TMs are powerful enough to:
  1. Receive other TMs as input
  2. Construct other TMs
  3. Simulate other TMs

- That's enough to achieve recursion!

A ***Turing machine*** is a 7-tuple, $(Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject})$, where Q, Σ, Γ are all finite sets and

1. Q is the set of states,
2. Σ is the input alphabet not containing the ***blank symbol*** $\sqcup$,
3. Γ is the tape alphabet, where $\sqcup \in \Gamma$ and $\Sigma \subseteq \Gamma$,
4. $\delta \colon Q \times \Gamma \longrightarrow Q \times \Gamma \times \{L, R\}$ is the transition function,
5. $q_0 \in Q$ is the start state,
6. $q_{accept} \in Q$ is the accept state, and
7. $q_{reject} \in Q$ is the reject state, where $q_{reject} \neq q_{accept}$.

Where's the recursion???

Check-in Quiz 11/1

On gradescope