Turing Machines and Recursion

Monday, April 11, 2022
Announcements

• HW 9
  • due Sun 4/17 11:59pm EST

• No lecture next Monday 4/18
Recursion in Programming

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

Most programming languages allow a function to call itself **recursively**, even before it’s completely defined!
Live Coding: Recursive Functions

- **Recursion**: typically “built into” a programming language

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

Next:
- Does recursion need to be a “built into” the language?
- E.g., Could you write this same function ...
  ... in a language without explicit recursion?
Turing Machines and Recursion

We’ve been saying: “A Turing machine models programs.”

**Q:** Is a recursive program modeled by a Turing machine?

**A:** Yes!
- But it’s not explicit.
- In fact, it’s a little complicated.
- Need to prove it...

**Today:** The Recursion Theorem

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

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

Where’s the recursion in this definition???
The Recursion Theorem

• You can write a TM description like this:

\[ B = \text{“On input } w: \]
\[ \quad \text{1. Obtain, via the recursion theorem, own description } \langle B \rangle. \]
The Recursion Theorem

Prove $A_{TM}$ is undecidable, by contradiction:
assume that Turing machine $H$ decides $A_{TM}$

$B = \text{"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.”

Example Use Case

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

This is the impossible “$D$” machine, it 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?
Where “Recursion” Comes From

TMs:
1. Have a string representation
2. Can simulate other TMs
3. Can receive other TMs as input

So to simulate recursion ...
... add an extra input and assume it will be copy of yourself!
A Simpler Exercise

Our Task:

• Create a TM that, without using recursion, prints itself.
  • How does this TM get knowledge about “itself”?

• An example, in English:

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

• This TM knows about “itself”,
  • but it does not explicitly use recursion!

Idea:
TMs can receive TMs as input;
Just assume input will be yourself!
(because a TM definition, like a program, is just a string)
Live Coding: Self-Printing Program

Our Task:
• Create a program that, without using recursion, prints itself.

• An example, in English:

  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”
  (program gets itself from its input!)
Interlude: Lambda

- $\lambda =$ anonymous function, e.g. $(\lambda x) x$
  - **C++**: [](int x){ return x; }
  - **Java**: (x) -> { return x; }
  - **Python**: lambda x : x
  - **JS**: (x) => { return x; }
Live Coding: Self-Printing Program

Our Task:
• Create a program that, **without using recursion**, prints itself.

• An example, in English:

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

  “input”

  “argument”
  (program gets itself from its input!)

  “itself”
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:”

```
(define (print2x str)
  (printf "\n\n" str str))

((\ (SELF) (print2x SELF))
  "((\ (SELF) (print2x SELF))")
)
```

First copy

Second copy (quoted)

(could have inlined this)
Self-Printing Turing Machine

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. Erase input.  \[1\]
   2. Write $w$ on the tape.
   3. Halt.”

2. Output $\langle P_w \rangle$.” \[2\]

$q$ 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” $M$

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

“Print out two copies of the following, the second on in quotes:”
**SELF, Defined With The Recursion Theorem**

\[ SELF = \text{"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”?

Could we write a recursive program that does **something other than print “itself”**?
A Recursion Code Example

(define (factorial n) ;; R
  (if (zero? n)
    1
    (* n (factorial (sub1 n))))
The Recursion Theorem, Formally

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

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

In English:

- 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 $T$ ???
Recursion Theorem, A Code Example

- If you want:
  - Recursive fn

- Instead create:
  - Non-recursive fn

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

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

Recursion Theorem says: can always convert 2nd one to 1st one

But how???
The Recursion Theorem, Pictorially

• To convert a “(T)” to “(R)”:

\[
A \rightarrow B \rightarrow T \quad (=P_{\langle BT \rangle})
\]

control for \(R\)

\[
\quad AB = SELF \text{(prev slide)}
\]
\[
T = \text{machine that gets}
\]
\[
SELF \text{ as argument}
\]
\[
R = T \text{ without explicit}
\]
\[
SELF \text{ argument}
\]

1. Construct \(A = \text{program constructing } \langle BT \rangle\), and
2. Pass result to \(B \) (from before),
3. which passes “itself” to \(T\)
Non-Printing Uses of \textit{SELF}

- Program that prints “itself”:

\[
((\lambda (\text{SELF}) (\text{print2x SELF})) \\
(\lambda (\text{SELF}) (\text{print2x SELF})))
\]

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

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

  Call arg fn with itself as arg

  Don’t convert arg to string

- Loop, but do something useful each time?

\[
((\lambda (\text{SELF}) (f (\text{SELF SELF}))) \\
(\lambda (\text{SELF}) (f (\text{SELF SELF}))))
\]

\[
((\lambda (\text{SELF}) (f (\lambda (v) ((\text{LEFT SELF SELF} v)))))) \\
(\lambda (\text{SELF}) (f (\lambda (v) ((\text{LEFT SELF SELF} v))))))
\]

- None of these programs use explicit recursion!

\textbf{Y combinator}
Recursion Theorem Proof: Coding Demo

• Program that passes “itself” to another function:

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

Y combinator

Pass to

• Function that needs “itself”

\[
\text{(define (factorial/itself ITSELF n) ;; T (if (zero? n) 1 (* n (ITSELF (sub1 n)))))}
\]

Y combinator is the “converter” guaranteed by the Recursion Theorem!
Fixed Points

• A value $x$ is a fixed point of a function $f$ if $f(x) = x$
Recursion Theorem and Fixed Points

Let $t: \Sigma^* \rightarrow \Sigma^*$ 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 $G$.

• **l.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

- \texttt{mk-recursive-fn} = a “fixed point finder”

\begin{verbatim}
(define mk-recursive-fn
  (\lambda (f)
    (((\lambda (x) (f (\lambda (v) ((x x) v))))
      (\lambda (x) (f (\lambda (v) ((x x) v))))))))
\end{verbatim}

- \texttt{factorial} is the fixed point of \texttt{mk-factorial}
Summary: Where “Recursion” Comes From

- TMs:
  1. Have a string representation
  2. Can receive other TMs as input
  3. Can simulate other TMs

- That’s enough to achieve recursion!

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

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

Where’s the recursion???
Check-in Quiz 4/11
On gradescope