Automata Theory
CS411-2015F-14

Counter Machines & Recursive Functions

David Galles
Department of Computer Science
University of San Francisco
• Give a Non-Deterministic Finite Automata a counter
  • Increment the counter
  • Decrement the counter
  • Check to see if the counter is zero
A Counter Machine $M = (K, \Sigma, \Delta, s, F)$
- $K$ is a set of states
- $\Sigma$ is the input alphabet
- $s \in K$ is the start state
- $F \subset K$ are Final states
- $\Delta \subseteq ((K \times (\Sigma \cup \epsilon) \times \{\text{zero, } \neg\text{zero}\}) \times (K \times \{-1, 0, +1\}))$

Accept if you reach the end of the string, end in an accept state, and have an empty counter.
14-2: Counter Machines

- Give a Non-Deterministic Finite Automata a counter
  - Increment the counter
  - Decrement the counter
  - Check to see if the counter is zero
- Do we have more power than a standard NFA?
Give a counter machine for the language $a^n b^n$
Give a counter machine for the language $a^n b^n$

\[(a, \text{zero}, +1)\]
\[(a, \sim\text{zero}, +1)\]
\[(b, \sim\text{zero}, -1)\]
Give a 2-counter machine for the language $a^n b^n c^n$

- Straightforward extension – examine (and change) two counters instead of one.
Give a 2-counter machine for the language $a^n b^n c^n$

- $(a, \text{zero, zero, +1, 0})$
- $(a, \text{~zero, zero, +1, 0})$
- $(b, \text{~zero, ~zero, -1, +1})$
- $(b, \text{~zero, zero, -1, +1})$
- $(c, \text{zero, ~zero, 0, -1})$
- $(c, \text{zero, ~zero, 0, -1})$
Our counter machines only accept if the counter is zero

Does this give us any more power than a counter machine that accepts whenever the end of the string is reached in an accept state?

That is, given a counter machine $M$ that accepts only strings that both drive the machine to an accept state, and leave the counter empty, can we create a counter machine $M'$ that accepts all strings that drive the machine to an accept state (regardless of the contents of the counter) so that $L[M] = L[M']$?
Our counter machines only accept if the counter is zero.

Does this give us any *more* power than a counter machine that accepts whenever the end of the string is reached in an accept state?
Our counter machines only accept if the counter is zero

Does this give us any *more* power than a counter machine that accepts whenever the end of the string is reached in an accept state?

\[(\varepsilon, \text{zero}, 0)\]
Our counter machines only accept if the counter is zero
- Does this give us any less power than a counter machine that accepts whenever the end of the string is reached in an accept state?
- That is, given a counter machine $M$ that accepts all strings that drive the machine to an accept state (regardless of contents of counter), can we create a counter machine $M'$ that accepts only strings that both drive the machine to an accept state and leave the counter empty, such that $L[M] = L[M']$?
14-11: **Counter Machines**

- Our counter machines only accept if the counter is zero
  - Does this give us any *less* power than a counter machine that accepts whenever the end of the string is reached in an accept state?
Counter Machines

- Our counter machines only accept if the counter is zero.
- Does this give us any less power than a counter machine that accepts whenever the end of the string is reached in an accept state?
14-13: **Counter Machines**

- Give a Non-Deterministic Finite Automata *two* counters
- We can use two counters to simulate a stack
  - How?
  - *HINT*: We will simulate a stack that has two symbols, 0 and 1
  - *HINT₂*: Think binary
We can use two counters to simulate a stack:

- One counter will store the contents of the stack.
- Other counter will be used as “scratch space.”

Stack will be represented as a binary number, with the top of the stack being the least significant bit:

- How can we push a 0?
- How can we push a 1?
14-15: **Counter Machines**

- How can we push a 0?
  - Multiply the counter by 2
- How can we push a 1?
  - Multiply the counter by 2, and add 1
How can we multiply a counter by 2, if all we can do is increment?

- Remember, we have a “scratch counter”
Counter Machines

- How can we multiply a counter by 2, if all we can do is increment:
  - Set the “Scratch Counter” to 0
  - While counter is not zero:
    - Decrement the counter
    - Increment the “Scratch Counter” twice
To Push a 0:
- While Counter1 \( \neq 0 \)
  - Increment Counter2
  - Increment Counter2
  - Decrement Counter1
- Swap Counter1 and Counter2
To Push a 1:

- While Counter1 $\neq 0$
  - Increment Counter2
  - Increment Counter2
  - Decrement Counter1
- Increment Counter2
- Swap Counter1 and Counter2
14-20: **Counter Machines**

- **To Pop:**
  - While $\text{Counter1} \neq 0$
    - Decrement Counter1
    - If $\text{Counter1} = 0$, popped result is 1
    - Decrement Counter1
    - If $\text{Counter1} = 0$, popped result is 0
    - Increment Counter2
  - Swap Counter1 and Counter2
How do we check if the simulated stack is empty?

- We need to use 1 (not zero) to represent an empty stack (why?)
- Stack is empty if \((\text{counter1} - 1 = 0)\)
14-22: **Counter Machines**

- **Example**

  Stack Counter  | Scratch Counter
  ---------------|-----------------|
  1              | 0

- Stack counter starts out as 1 (represents empty stack)
- Scratch counter starts out as 0
Counter Machines

- Example

Stack Counter: 1
Scratch Counter: 0

- Push 0
Example

Stack Counter | Scratch Counter
---|---
0 | 1

Decrement Stack Counter, increment scratch counter
14-25: **Counter Machines**

- **Example**
  
  Stack Counter: 0  
  Scratch Counter: 10

- **Decrement Stack Counter, increment scratch counter (twice)**
Example

Stack Counter | Scratch Counter
---|---
0 | 10

Swap Scratch Counter and Stack Counter

While Scratch Counter $\neq$ Stack Counter
Decrement Scratch Counter
Increment Stack Counter
Counter Machines

Example

Stack Counter

\[10\]

Scratch Counter

\[0\]

Swap Scratch Counter and Stack Counter

While Scratch Counter \(\neq\) Stack Counter
Decrement Scratch Counter
Increment Stack Counter
Example

Stack Counter: 10
Scratch Counter: 0

Push 1
Example

Stack Counter | Scratch Counter
---|---
1 | 1

- Decrement Stack Counter, increment scratch counter
Counter Machines

- Example

<table>
<thead>
<tr>
<th>Stack Counter</th>
<th>Scratch Counter</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10</td>
</tr>
</tbody>
</table>

- Decrement Stack Counter, increment scratch counter (twice)
### 14-31: Counter Machines

- **Example**

<table>
<thead>
<tr>
<th>Stack Counter</th>
<th>Scratch Counter</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>11</td>
</tr>
</tbody>
</table>

- Decrement Stack Counter, increment scratch counter
14-32: Counter Machines

- Example

<table>
<thead>
<tr>
<th>Stack Counter</th>
<th>Scratch Counter</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>100</td>
</tr>
</tbody>
</table>

- Decrement Stack Counter, increment scratch counter (twice)
### Example

<table>
<thead>
<tr>
<th>Stack Counter</th>
<th>Scratch Counter</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>101</td>
</tr>
</tbody>
</table>

- Add one to scratch counter (since pushing 1, not 0)
14-34: Counter Machines

- Example

<table>
<thead>
<tr>
<th>Stack Counter</th>
<th>Scratch Counter</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>101</td>
</tr>
</tbody>
</table>

- Swap Scratch Counter and Stack Counter

While Scratch Counter $\neq$ Stack Counter
Decrement Scratch Counter
Increment Stack Counter
14-35: Counter Machines

- Example

<table>
<thead>
<tr>
<th>Stack Counter</th>
<th>Scratch Counter</th>
</tr>
</thead>
<tbody>
<tr>
<td>101</td>
<td>0</td>
</tr>
</tbody>
</table>

- Swap Scratch Counter and Stack Counter

While Scratch Counter ≠ Stack Counter
Decrement Scratch Counter
Increment Stack Counter
14-36: **Counter Machines**

- **Example**

<table>
<thead>
<tr>
<th>Stack Counter</th>
<th>Scratch Counter</th>
</tr>
</thead>
<tbody>
<tr>
<td>101</td>
<td>0</td>
</tr>
</tbody>
</table>

- **Pop**
14-37: Counter Machines

- Example

<table>
<thead>
<tr>
<th>Stack Counter</th>
<th>Scratch Counter</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>0</td>
</tr>
</tbody>
</table>

- Decrement Stack counter
Example

Stack Counter | Scratch Counter
---|---
11 | 0

Decrement Stack counter (twice)
**Counter Machines**

- Example

<table>
<thead>
<tr>
<th>Stack Counter</th>
<th>Scratch Counter</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>1</td>
</tr>
</tbody>
</table>

- Increment Scratch counter
### Counter Machines

- **Example**

<table>
<thead>
<tr>
<th>Stack Counter</th>
<th>Scratch Counter</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>1</td>
</tr>
</tbody>
</table>

- Decrement Stack counter
Counter Machines

- Example

Stack Counter | Scratch Counter
---|---
1 | 1

- Decrement Stack counter (twice)
Example

Stack Counter: 1
Scratch Counter: 10

Increment Scratch counter
14-43: **Counter Machines**

- Example

<table>
<thead>
<tr>
<th>Stack Counter</th>
<th>Scratch Counter</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>10</td>
</tr>
</tbody>
</table>

- Decrement Stack counter
**Example**

Stack Counter | Scratch Counter
--- | ---
0 | 10

• Can’t Decrement Stack counter a second time (empty), so popped value is 1
14-45: Counter Machines

• Example

Stack Counter  Scratch Counter
0 10

• Swap Scratch Counter and Stack Counter

While Scratch Counter ≠ Stack Counter
Decrement Scratch Counter
Increment Stack Counter
• Example

<table>
<thead>
<tr>
<th>Stack Counter</th>
<th>Scratch Counter</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>0</td>
</tr>
</tbody>
</table>

• Swap Scratch Counter and Stack Counter

While Scratch Counter $\neq$ Stack Counter
Decrement Scratch Counter
Increment Stack Counter
Two counters can simulate a stack
Four counters can simulate two stacks
What can we do with two stacks?
Counter Machines

- Two stacks can simulate a Turing Machine:
  - Stack 1: Everything to the left of the read/write head
  - Stack 2: Everything to the right of the read/write head
- Tape head points to top of Stack 1
Counter Machines

- To write a new symbol at the Tape Head
- Pop old value off the top of Stack 1
- Push new value on the top of Stack 1

Turing Machine

Stack 1

Stack 2
14-50: Counter Machines

- To write a new symbol at the Tape Head
- Pop old value off the top of Stack 1
- Push new value on the top of Stack 1

Turing Machine

Stack 1  Stack 2
Counter Machines

- To move the tape head to the left
  - Pop symbol off Stack 1
  - Push it on Stack 2

Turing Machine

Stack 1 Stack 2

- Stack 1
  - X
  - b
  - a
- Stack 2
  - d
  - e
  - f
  - g
14-52: Counter Machines

- To move the tape head to the left
  - Pop symbol off Stack 1
  - Push it on Stack 2

Turing Machine

Stack 1

Stack 2
**14-53: Counter Machines**

- To move the tape head to the right
  - Pop symbol off Stack 2
  - Push it on Stack 1

---

**Turing Machine**

```
D a b X d e f g ...
```

Stack 1

```
 b a
```

Stack 2

```
X
d
e
f
```
14-54: Counter Machines

- To move the tape head to the right
- Pop symbol off Stack 2
- Push it on Stack 1

Turing Machine

Stack 1

Stack 2
To move the tape head to the right, if Stack 2 is empty ...
To move the tape head to the right, if Stack 2 is empty ...

- Push a Blank Symbol on Stack 1
14-57: Counter Machines

- To move the tape head to the right, if Stack 2 is empty ...
- Push a Blank Symbol on Stack 1
Counter Machines

- Four Counters $\Rightarrow$ Two Stacks $\Rightarrow$ Turing Machine
- If we can simulate a 4-counter machine with a 2-counter machine ...
- Two Counters $\Rightarrow$ Four Counters $\Rightarrow$ Two Stacks $\Rightarrow$ Turing Machine
• We can represent 4 counters using just one counter

• Counters have values $i, j, k, l$

• Master Counter: $2^i 3^j 5^k 7^l$

• When all counters have value 0, master counter has value 1
Master Counter: \(2^i 3^j 5^k 7^l\)

- To increment counter \(j\), multiply Master Counter by 3
  - Decrement Master Counter
  - Increment Scratch Counter 3 times
  - Repeat until Master Counter = 0
  - Move Scratch Counter to Master Counter
Master Counter: $2^i 3^j 5^k 7^l$

To decrement counter $j$, divide Master Counter by 3
- Decrement Master Counter 3 times
- Increment Scratch Counter
- Repeat until Master Counter $= 0$
- Copy Scratch Counter to Master Counter
14-62: 2 Counter ⇒ 4 Counter

- Master Counter: \(2^i 3^j 5^k 7^l\)
- To check if counter \(j\) is zero, see if MC mod 3 = 0
  - Decrement Master Counter 3 times (if we hit zero in the middle of this operation, MC mod 3 ≠ 0, if hit zero at the end, MC mod 3 = 0)
  - Increment Scratch Counter 3 times
  - Repeat until Master Counter = 0
  - Use Scratch Counter to restore Master Counter
Counter Machines

- Machine with:
  - Finite State Control
  - Two counters
    - Increment, Decrement, check for zero
- Has full power of a Turing machine – can compute anything
• New model of computation: Recursive Functions
  • Very simple functions
  • Method of combining functions
• End up with equivalent power of Turing Machines
Basic Functions:

- Zero function: \( \text{zero}_k(n_1, \ldots, n_k) = 0 \)
- Identity function: \( \text{id}_{k,j}(n_1, \ldots, n_k) = n_j \)
- Successor function: \( \text{succ}(n) = n + 1 \) for all \( n \in \mathbb{N} \)
• Zero Function:
  • \( zero_3(3, 11, 22) = 0 \)
  • \( zero_2(9, 13) = 0 \)
  • \( zero_0() = 0 \)
Zero Function:

- Why have \( k \)-ary zero function, instead of just defining a the constant 0, or a single 0-ary function?
  - Notational convenience
  - “Historical Reasons”
Identity function

• $id_{1,1}(4) = 4$
• $id_{4,2}(3, 7, 9, 5) = 7$
• $id_{5,5}(9, 11, 4, 5, 20) = 20$
Successor Function

- $\text{succ}(0) = 1$
- $\text{succ}(1) = 2$
- $\text{succ}(2) = 3$
- $\text{succ}(57) = 58$
Combining Functions:

- Composition
  - $g : \mathbb{N}^k \mapsto \mathbb{N}$ any $k$-ary function
  - $h_1, \ldots, h_k$ $l$-ary functions
  - Composition of $g$ with $h_1, \ldots, h_k$

$$f(n_1, \ldots, n_l) = g(h_1(n_1, \ldots, n_l), \ldots, h_k(n_1, \ldots, n_l))$$
Composition:

- \( plus2(x) = succ(succ(x)) \)
- \( plus3(x) = succ(succ(succ(x))) \)
Composition: Constant functions

- \( f() = 5 = \text{succ}(\text{succ}(\text{succ}(\text{succ}(\text{succ}(\text{zero}())))))) \)
- \( f(3, 2) = 2 = \text{succ}(\text{succ}(\text{zero}())) \)
Combining Functions:

- Recursion
  - \(k\)-ary function \(g\), \(k + 2\)-ary function \(h\)
  - Function \(f\) defined recursively by \(g\) and \(h\):

\[
\begin{align*}
  f(n_1, \ldots n_k, 0) &= g(n_1, \ldots, n_k) \\
  f(n_1, \ldots, n_k, m + 1) &= h(n_1, \ldots n_k, m, f(n_1, \ldots, n_k, m))
\end{align*}
\]
Recursive functions:

\[
\begin{align*}
\text{plus}(m, 0) &= m \\
\text{plus}(m, n + 1) &= \text{succ}(\text{plus}(m, n))
\end{align*}
\]
Recursive functions:

\[
\begin{align*}
\text{plus}(m, 0) &= m \\
\text{plus}(m, n + 1) &= \text{succ}(\text{plus}(m, n))
\end{align*}
\]

\[
\begin{align*}
g(n) &= \text{id}_{1,1}(n) = n \\
h(n_1, n_2, n_3) &= \text{succ}(n_3)
\end{align*}
\]
Recursive functions:

\[\text{mult}(m, 0) = \]
\[\text{mult}(m, n + 1) = \]
Recursive functions:

\[
\begin{align*}
\text{mult}(m, 0) & = \text{zero}(m) \\
\text{mult}(m, n + 1) & = \text{plus}(m, \text{mult}(m, n))
\end{align*}
\]
Recursive functions:

\[
\text{mult}(m, 0) = \text{zero}(m) \\
\text{mult}(m, n + 1) = \text{plus}(m, \text{mult}(m, n))
\]

- \( g(n) = \text{zero}(n) \)
- \( h(n_1, n_2, n_3) = \text{plus}(n_1, n_3) \)
Recursive functions:

\[ \exp(m, 0) = \]
\[ \exp(m, n + 1) = \]
Recursive functions:

\[ \begin{align*}
    \text{exp}(m, 0) &= \text{suc}(\text{zero}(m)) \\
    \text{exp}(m, n + 1) &= \text{mult}(m, \text{exp}(m, n))
\end{align*} \]

- \( g(n) = \text{succ}(\text{zero}(n)) \)
- \( h(n_1, n_2, n_3) = \text{mult}(n_1, n_3) \)
Recursive functions:

\[
\begin{align*}
\text{fact}(0) &= \quad \\
\text{fact}(n + 1) &= \\
\end{align*}
\]
Recursive functions:

\[ \text{fact}(0) = \text{suc}(\text{zero}()) \]
\[ \text{fact}(n + 1) = \text{mult}(n + 1, \text{fact}(n)) \]

- \( g(n) = \text{succ}(\text{zero}(n)) \)
- \( h(n_1, n_2) = \text{mult}(\text{succ}(n_1), n_2) \)
Recursive functions:

\[ \text{pred}(0) = 0 \]
\[ \text{pred}(n + 1) = n \]

- \( g(n) = \text{zero}(n) \)
- \( h(n_1, n_2) = \text{id}_{12}(n_1, n_2) = n_1 \)
Recursive functions:

\[
\begin{align*}
\text{sub}(m, 0) &= m \\
\text{sub}(m, n + 1) &= \text{pred}(\text{sub}(m, n))
\end{align*}
\]

What is \(\text{sub}(3, 5)\)? Why?
Predicate functions

- \( iszero(n) = 1 \) if \( n = 0 \), and 0 otherwise

\[
\begin{align*}
  iszero(0) &= 1 \\
  iszero(m + 1) &= 0
\end{align*}
\]
**Predicate functions**

- $geq(m, n) = 1$ if $m \geq n$, and 0 otherwise

$$geq(m, n) =$$
Predicate functions

\[ \text{geq}(m, n) = 1 \text{ if } m \geq n, \text{ and 0 otherwise} \]

\[ \text{geq}(m, n) = \text{iszero}(\text{sub}(n, m)) \]
Predicate functions

\[ lt(m, n) = 1 \text{ if } m < n, \text{ and } 0 \text{ otherwise} \]
Numerical Functions

• Predicate functions
  • \( \text{lt}(m, n) = 1 \) if \( m < n \), and 0 otherwise

\[
\text{lt}(m, n) = \text{sub}(1, \text{geq}(m, n))
\]
Predicate functions

\[ \text{and}(m, n) = 1 \text{ if } m = 1 \text{ and } n = 1, \text{ and } 0 \text{ otherwise} \]
Predicate functions

\[ \text{and}(m, n) = 1 \text{ if } m = 1 \text{ and } n = 1, \text{ and } 0 \text{ otherwise} \]

\[ \text{and}(m, n) = \text{mult}(m, n) \]
14-92: Numerical Functions

- Predicate functions
  - \( \text{or}(m, n) = 1 \) if \( m = 1 \) or \( n = 1 \), and 0 otherwise
**Predicate functions**

- \( \text{or}(m, n) = 1 \) if \( m = 1 \) or \( n = 1 \), and 0 otherwise

\[
\text{or}(m, n) = \text{sub}(1, \text{iszero}(\text{plus}(m, n)))
\]
Defining functions by cases:

\[
f(n_1, \ldots, n_k) = \begin{cases} 
  g(n_1, \ldots, n_k) & \text{if } p(n_1, \ldots, n_k) \\
  h(n_1, \ldots, n_k) & \text{otherwise}
\end{cases}
\]
Defining functions by cases:

\[ \text{rem}(0, n) = 0 \]

\[ \text{rem}(m + 1, n) = \begin{cases} 
0 & \text{if } \text{equal} (\text{rem}(m, n), \text{pred}(n)) \\
\text{rem}(m, n) + 1 & \text{otherwise}
\end{cases} \]

(Using first parameter of function as recursion control)
Defining functions by cases:

\[
\begin{align*}
\text{div}(0, n) &= 0 \\
\text{div}(m + 1, n) &= \begin{cases} 
\text{div}(m, n) + 1 & \text{if } \text{equal}(\text{rem}(m, n), \text{pred}(n)) \\
\text{div}(m, n) & \text{otherwise}
\end{cases}
\end{align*}
\]

(Using first parameter of function as recursion control)
Defining functions by cases:

$$f(n_1, n_2, \ldots, n_k) = \begin{cases} g(n_1, n_2, \ldots, n_k) & \text{if } P(n_1, n_2, \ldots, n_k) \\ h(n_1, n_2, \ldots, n_k) & \text{otherwise} \end{cases}$$

How can we get “functions by cases” using the tools we already have?
14-98: **Numerical Functions**

- Defining functions by cases:

\[
f(n_1, n_2, \ldots, n_k) = \begin{cases} 
g(n_1, n_2, \ldots, n_k) & \text{if } P(n_1, n_2, \ldots, n_k) \\
h(n_1, n_2, \ldots, n_k) & \text{otherwise} \end{cases}
\]

\[
f(n_1, n_2, \ldots, n_k) = P(n_1, n_2, \ldots, n_k) \ast g(n_1, n_2, \ldots, n_k) \\
+ ((1 - P(n_1, n_2, \ldots, n_k)) \ast h(n_1, n_2, \ldots, n_k))
\]
Are there any functions which we can compute, that \textit{cannot} be computed with primitive recursive functions?
Are there any functions which we can compute, that cannot be computed with primitive recursive functions?

- Yes!
  - Use a diagonalization argument

- To make life easier, we will only consider functions that take a single argument (unary functions)
Numerical Functions

• Unary Primitive Recursive Functions can be enumerated
  • That is, we can define an order over all unary primitive recursive functions,
    \( f_1(n), f_2(n), f_3(n), \ldots \)
  • How can we order them?
Enumerating Unary Primitive Recursive Functions

- Each function is created by combining basic functions (succ, zero, select, etc) using composition and recursion
- Can describe any function using a string
- Order the strings in lexicographic order (shortest to longest, using standard string compare for strings of the same length)
Let the unary primitive recursive functions be: 
\[ f_0, f_1, f_2, f_3, \ldots \]

Define a new function \( g(n) = f_n(n) + 1 \)

- We can compute \( g(n) \) by first finding the \( n \)th unary recursive function \( f_n \), computing \( f_n(n) \), and adding 1 to the result.
Let the unary primitive recursive functions be: $f_0, f_1, f_2, f_3, \ldots$

Define a new function $g(n) = f_n(n) + 1$

- We can compute $g(n)$ by first finding the $n$th unary recursive function $f_n$, computing $f_n(n)$, and adding 1 to the result.

$g(n)$ can be computed (we just showed how).

$g(n)$ cannot be computed by a primitive recursive function! (why not?)
Numerical Functions

- $g(n)$ can be computed (we just showed how)
- $g(n)$ cannot be computed by a primitive recursive function! (why not?)
  - Not computed by the 0th primitive recursive function
  - Not computed by the 1st primitive recursive function
  - Not computed by the 2nd primitive recursive function
  - ...
There are some well defined functions, which we can compute, which cannot be computed by primitive recursive functions.

Can we add anything to primitive recursive functions to give them more power, so that any well defined function that can be computed can be computed with recursive functions?
Minimization

If \( g \) is a \((k+1)\)-ary function. The minimalization of \( g \) is the \( k \)-ary function \( f \) defined as:

\[
f(n_1, \ldots, n_k) = \begin{cases} 
\text{The least } m \text{ such that } g(n_1, \ldots, n_k, m) = 1, \\
\text{if such an } m \text{ exists} \\
0 \text{ otherwise}
\end{cases}
\]

Minimization of \( g \) is denoted \( \mu_m[g(n_1, \ldots n_k, m) = 1] \)
Minimization Examples

\[ \text{div}(x, y) = \mu z [(y \ast (z + 1)) - x > 0] \]

“−” is “positive subtraction” (that is, if \( y > x \), then \( x - y = 0 \))

\[ \text{div}(x, y) = z \]
\[ y \ast z \leq x \]
\[ y \ast (z + 1) > x \]
Minimization Examples

\[ \log(m, n) = \mu_p[\text{power}(m, p) \geq n] \]

“\( \geq \)” is the “greater-than-or-equal” predicate
Calculating minimalization:

\[ m \leftarrow 0; \]
\[ \text{while } (g(n_1, \ldots, n_k, m) \neq 1) \]
\[ \quad m \leftarrow m + 1 \]
\[ \text{return } m \]
Calculating minimalization:

\[ m \leftarrow 0; \]
\[ \text{while } (g(n_1, \ldots, n_k, m) \neq 1) \]
\[ m \leftarrow m + 1 \]

return \( m \)

... of course, this may never terminate, if there is no value of \( m \) such that \( g(n_1, \ldots, n_k, m) = 1 \)
A function \( g(n_1, \ldots, n_k, m) \) is minimalizable if

- For each \( n_1, \ldots, n_k \in \mathbb{N} \), there exists some \( m \) such that \( g(n_1, \ldots, n_k, m) = 1 \)

That is:

\[
\begin{align*}
m &\leftarrow 0; \\
\text{while } & (g(n_1, \ldots, n_k, m) \neq 1) \\
& \quad m \leftarrow m + 1 \\
\text{return } & m
\end{align*}
\]

always terminates, for all values \( n_1, \ldots, n_k \)
**μ-Recursive**

- A function is $\mu$-recursive if it consists entirely of primitive-recursive functions, and minimalizations of minimalizable functions.

- $\mu$-recursive functions can calculate anything that can be decided by a Turing machine.
  - (recall that “decide” means the TM halts on all inputs)
• \( \mu \)-recursive functions can calculate anything that can be decided by a Turing machine

• We can enumerate \( \mu \)-recursive functions just like we enumerated primitive recursive functions \( f_0, f_1, f_2, \ldots \)

• We can define the function \( g(n) = f_n(n) + 1 \)

• How can I assert that \( \mu \)-recursive functions can compute anything that a Turing Machine can compute, when \( \mu \)-recursive functions can’t compute \( g \)?
Method to compute $g(n)$ using a Turing machine:

- Enumerate first $n + 1$ functions $f$
  - $f_0, f_1, \ldots, f_n$
- Compute $f_n(n)$
- Output $f_n(n) + 1$
Numerical Functions

- Method to compute \( g(n) \) using a Turing machine:
  - Enumerate first \( n + 1 \) functions \( f \)
    - \( f_0, f_1, \ldots, f_n \)
  - Compute \( f_n(n) \)
  - Output \( f_n(n) + 1 \)

- Function \( f_n \) might not be minimalizable! If \( f_n(n) \) is not minimalizable, then \( f_n(n) = 0 \), but we have no way of discovering this!
Recursive Languages

- $\mu$-recursive functions can calculate anything that can be decided by a Turing machine.
- $\{L : L \text{ is decided by some TM } M\}$ is the recursive languages
- How can a function from the natural numbers to the natural numbers decide a language?
Recursive Languages

- How can a function from the natural numbers to the natural numbers decide a language?
  - Any string can be encoded as a number
    - ASCII-style encoding scheme to encode each symbol in string
    - Append codes of each symbol together to get a (really large) number

\[ \Sigma = \{a, \ldots, z\} \]

- \( en(a) = 10 \), \( en(b) = 11 \), \ldots,
- \( en(z) = 35 \)
- \( en(abbz) = 10111135 \)
Recursive Languages

How can a function from the natural numbers to the natural numbers decide a language?

- Any string can be encoded as a number
- Predicate function can be used to determine membership

\[ L[f] = \{w : f(en(w)) = 1\} \]
Any $\mu$-recursive function can be decided by a Turing machine

Each of the primitive-recursive functions can easily be simulated by a Turing machine

Any minimalizable function can be computed by a Turing machine that tries all values in order

```plaintext
m ← 0;
while (g(n_1, \ldots, n_k, m) \neq 1)
    m ← m + 1
return m
```
Any function that can be decided by a Turing machine can be computed with a μ-recursive function.

- We can encode a configuration as a number.
- We can encode a sequence of configurations with a (much larger) number.

Each configuration encodes tape contents, head location, and current state of the Turing Machine.
We have a large number, which represents a series of configurations for a Turning Machine \( \text{config}_1 \text{config}_2 \text{config}_3 \ldots \text{config}_n \).

We can write a primitive-recursive predicate function \( \text{isValid} \) that examines this string of configurations, and determines if it is legal:
- if \( \text{config}_i \text{config}_j \) appears in the sequence
- Turing machine will move from \( \text{config}_i \) to \( \text{config}_j \) in a single step
isvalid(n)

- Predicate function
- True if \( n \) is a number which represents a valid sequence of configurations of a Turing Machine
- Writing isvalid for a particular Turing Machine is reasonably straightforward
  - Extract 1st and 2nd configurations from the number (using div and mod)
  - Make sure that the transition from 1st to 2nd configuration is valid
  - Recursively check the rest of the transitions
Given a number which represents a valid sequence of configurations for the Turning Machine $M$, if:

- If the first configuration represents the initial state and the input $n$
- The last configuration contains a halting state $h$
- The tape contents of the last configuration represents the value $y$

Then the Turing Machine $M$ gives the output $y$ for the input $n$
Given a number which represents a sequence of configurations, and an input \( n \), we can:

- Determine if the sequence of configurations is valid
- Ensure that the first configuration encodes \( n \)
- Ensure that the last configuration contains a halting state
Function \textit{check\_compute}(n, x)

- Takes as input a string of configurations \( n \), and an initial configuration \( x \)
- Returns 1 (true) if \( n \) is a valid series of computations that starts with \( x \)
  - \( \text{isValid}(n) = 1 \)
  - \( \text{first}(n) = x \)
• Function $compute(x)$
  • Calculates and returns the string of valid configurations that starts with $x$ and ends in a halting state
• Function $compute(x)$
  • Calculates and returns the string of valid configurations that starts with $x$ and ends in a halting state

$$compute(x) = \mu n[check\_compute(n, x)]$$
Function $f_M$, that calculates the same function as the Turing Machine $M$:

$$f_M(x) = \text{last}(\text{compute}(x))$$