Programming Language Concepts: Lecture 19

S P Suresh

March 24, 2021

The extent of recursive functions

- For every recursive function $f : \mathbb{N}^k \to \mathbb{N}$ there is a λ-calculus expression $[f]$ such that

$$[f][n_1]\cdots[n_k] \xrightarrow{*}_\beta [f(n_1,\ldots,n_k)] \quad \text{for all } n_1,\ldots,n_k \in \mathbb{N}$$

The extent of recursive functions

- For every recursive function $f : \mathbb{N}^k \to \mathbb{N}$ there is a λ-calculus expression $[f]$ such that

$$[f][n_1]\cdots[n_k] \xrightarrow{*}_\beta [f(n_1,\ldots,n_k)] \quad \text{for all } n_1,\ldots,n_k \in \mathbb{N}$$

- Further if $[f][n_1]\cdots[n_k] \xrightarrow{*}_\beta [m]$ for any m, then $m = f(n_1,\ldots,n_k)$

The extent of recursive functions

- For every recursive function $f : \mathbb{N}^k \to \mathbb{N}$ there is a λ-calculus expression $[f]$ such that

$$[f][n_1]\cdots[n_k] \xrightarrow{*}_\beta [f(n_1,\ldots,n_k)] \quad \text{for all } n_1,\ldots,n_k \in \mathbb{N}$$

- Further if $[f][n_1]\cdots[n_k] \xrightarrow{*}_\beta [m]$ for any m, then $m = f(n_1,\ldots,n_k)$
- A consequence of the **Church-Rosser theorem**

The extent of recursive functions

- For every recursive function $f : \mathbb{N}^k \to \mathbb{N}$ there is a λ-calculus expression $[f]$ such that

$$[f][n_1]\cdots[n_k] \xrightarrow{*}_\beta [f(n_1,\ldots,n_k)] \quad \text{for all } n_1,\ldots,n_k \in \mathbb{N}$$

- Further if $[f][n_1]\cdots[n_k] \xrightarrow{*}_\beta [m]$ for any m, then $m = f(n_1,\ldots,n_k)$
- A consequence of the **Church-Rosser theorem**
- Thus all recursive functions can be expressed in the λ-calculus

The extent of recursive functions

- For every recursive function $f : \mathbb{N}^k \to \mathbb{N}$ there is a λ-calculus expression $[f]$ such that

$$[f][n_1]\cdots[n_k] \xrightarrow{*}_{\beta} [f(n_1, \ldots, n_k)] \quad \text{for all } n_1, \ldots, n_k \in \mathbb{N}$$

- Further if $[f][n_1]\cdots[n_k] \xrightarrow{*}_{\beta} [m]$ for any m, then $m = f(n_1, \ldots, n_k)$
- A consequence of the **Church-Rosser theorem**
- Thus all recursive functions can be expressed in the λ-calculus
- What functions are recursive? …

The extent of recursive functions

- For every recursive function $f : \mathbb{N}^k \to \mathbb{N}$ there is a λ-calculus expression $[f]$ such that

$$[f][n_1]\cdots[n_k] \xrightarrow{*}_\beta [f(n_1,\ldots,n_k)] \quad \text{for all } n_1,\ldots,n_k \in \mathbb{N}$$

- Further if $[f][n_1]\cdots[n_k] \xrightarrow{*}_\beta [m]$ for any m, then $m = f(n_1,\ldots,n_k)$
- A consequence of the **Church-Rosser theorem**
- Thus all recursive functions can be expressed in the λ-calculus
- What functions are recursive? …
- **Exactly the Turing computable functions!**

Recursive functions are computable

- We write programs for every recursive function

Recursive functions are computable

- We write programs for every recursive function
- **Initial functions**: Trivial programs

Recursive functions are computable

- We write programs for every recursive function
- **Initial functions**: Trivial programs
- **Composition**: If $f : \mathbb{N}^k \to \mathbb{N}$ is defined by $f = g \circ (h_1, \ldots, h_l)$

```
function f(x1, x2, ..., xk) {
    y1 = h1(x1, x2, ..., xk);
    y2 = h2(x1, x2, ..., xk);
    ...
    yl = hl(x1, x2, ..., xk);
    return g(y1, y2, ..., yl);
}
```

Recursive functions are computable

- **Primitive recursion** Suppose $f : \mathbb{N}^{k+1} \rightarrow \mathbb{N}$ is defined from $g : \mathbb{N}^k \rightarrow \mathbb{N}$ and $h : \mathbb{N}^{k+2} \rightarrow \mathbb{N}$ by

$$\begin{aligned} f(0, \vec{n}) &= g(\vec{n}) \\ f(i+1, \vec{n}) &= h(i, f(i, \vec{n}), \vec{n}) \end{aligned}$$

Recursive functions are computable

- **Primitive recursion** Suppose $f : \mathbb{N}^{k+1} \to \mathbb{N}$ is defined from $g : \mathbb{N}^k \to \mathbb{N}$ and $h : \mathbb{N}^{k+2} \to \mathbb{N}$ by

$$\begin{aligned} f(0, \vec{n}) &= g(\vec{n}) \\ f(i+1, \vec{n}) &= h(i, f(i, \vec{n}), \vec{n}) \end{aligned}$$

- Equivalent to computing a `for` loop:

```
result = g(n1, ..., nk);   // f(0, n1, ..., nk)
for (i = 0; i < n; i++) {  // computing f(i+1, n1, ..., nk)
    result = h(i, result, n1, ..., nk);
}
return result;
```

Recursive functions are computable

- μ-**recursion** Suppose $f : \mathbb{N}^k \rightarrow \mathbb{N}$ is defined from $g : \mathbb{N}^{k+1} \rightarrow \mathbb{N}$ by

$$f(\vec{n}) = \begin{cases} j & \text{if } g(j, \vec{n}) = 0 \text{ and } \forall i < j : g(i, \vec{n}) > 0 \\ \text{undefined} & \text{otherwise} \end{cases}$$

Recursive functions are computable

- *μ-recursion* Suppose $f : \mathbb{N}^k \to \mathbb{N}$ is defined from $g : \mathbb{N}^{k+1} \to \mathbb{N}$ by

$$f(\vec{n}) = \begin{cases} j & \text{if } g(j,\vec{n}) = 0 \text{ and } \forall i < j : g(i,\vec{n}) > 0 \\ \text{undefined} & \text{otherwise} \end{cases}$$

- Equivalent to computing a `while` loop:

```
i = 0;
while (g(i, n1, ..., nk) > 0) {i = i + 1;}
return i;
```

Some primitive recursive functions

- **Predecessor**

$$
\begin{aligned}
pred(0) &= Z(0) = 0 \\
pred(n+1) &= \Pi^2_1(n, pred(n)) = n
\end{aligned}
$$

Some primitive recursive functions

- **Predecessor**

$$pred(0) = Z(0) = 0$$
$$pred(n+1) = \Pi^2_1(n, pred(n)) = n$$

- **Integer difference**

$$x - 0 = x$$
$$x - (y+1) = pred(x - y)$$

Some primitive recursive functions

- **Predecessor**

$$pred(0) = Z(0) = 0$$
$$pred(n+1) = \Pi_1^2(n, pred(n)) = n$$

- **Integer difference**

$$x - 0 = x$$
$$x - (y+1) = pred(x-y)$$

- **Factorial**

$$0! = 1$$
$$(n+1)! = (n+1) \cdot n!$$

Some primitive recursive functions

- **Bounded sums** $g(z, \vec{x}) = \sum_{y \leq z} f(y, \vec{x})$

$$g(0, \vec{x}) = f(0, \vec{x})$$
$$g(y+1, \vec{x}) = g(y, \vec{x}) + f(y+1, \vec{x})$$

Some primitive recursive functions

- **Bounded sums** $g(z, \vec{x}) = \sum_{y \leq z} f(y, \vec{x})$

$$g(0, \vec{x}) = f(0, \vec{x})$$
$$g(y+1, \vec{x}) = g(y, \vec{x}) + f(y+1, \vec{x})$$

- **Bounded products** $g(z, \vec{x}) = \prod_{y \leq z} f(y, \vec{x})$

$$g(0, \vec{x}) = f(0, \vec{x})$$
$$g(y+1, \vec{x}) = g(y, \vec{x}) \cdot f(y+1, \vec{x})$$

Primitive recursive relations

- A relation $R \subseteq \mathbb{N}^k$ is primitive recursive if its characteristic function c_R is primitive recursive

Primitive recursive relations

- A relation $R \subseteq \mathbb{N}^k$ is primitive recursive if its characteristic function c_R is primitive recursive
- *iszero*

$$\begin{aligned} \mathit{iszero}(0) &= \mathit{true} & \qquad c_{\mathit{iszero}}(0) &= \mathit{succ}(\Pi_1^1(0)) \\ \mathit{iszero}(n+1) &= \mathit{false} & \qquad c_{\mathit{iszero}}(n+1) &= Z(n) \end{aligned}$$

Primitive recursive relations

- A relation $R \subseteq \mathbb{N}^k$ is primitive recursive if its characteristic function c_R is primitive recursive
- *iszero*

$$\begin{aligned} iszero(0) &= true & \qquad c_{iszero}(0) &= succ(\Pi_1^1(0)) \\ iszero(n+1) &= false & \qquad c_{iszero}(n+1) &= Z(n) \end{aligned}$$

- $x \leq y$ iff $iszero(x - y)$, so $c_{\leq}(x, y) = c_{iszero}(x - y)$

Primitive recursive relations

- A relation $R \subseteq \mathbb{N}^k$ is primitive recursive if its characteristic function c_R is primitive recursive
- *iszero*

$$
\begin{aligned}
\mathit{iszero}(0) &= \mathit{true} & \qquad c_{\mathit{iszero}}(0) &= \mathit{succ}(\Pi_1^1(0)) \\
\mathit{iszero}(n+1) &= \mathit{false} & \qquad c_{\mathit{iszero}}(n+1) &= Z(n)
\end{aligned}
$$

- $x \leq y$ iff $\mathit{iszero}(x-y)$, so $c_{\leq}(x,y) = c_{\mathit{iszero}}(x-y)$
- $c_{\neg P} = 1 - c_P$, $c_{P \wedge Q} = c_P \cdot c_Q$

Primitive recursive relations

- A relation $R \subseteq \mathbb{N}^k$ is primitive recursive if its characteristic function c_R is primitive recursive
- *iszero*

$$iszero(0) = true \qquad c_{iszero}(0) = succ(\Pi_1^1(0))$$

$$iszero(n+1) = false \qquad c_{iszero}(n+1) = Z(n)$$

- $x \leq y$ iff $iszero(x - y)$, so $c_{\leq}(x, y) = c_{iszero}(x - y)$
- $c_{\neg P} = 1 - c_P$, $c_{P \wedge Q} = c_P \cdot c_Q$
- For $Q(z, \vec{x}) = (\forall y \leq z) R(y, \vec{x})$, $c_Q(z, \vec{x}) = \prod_{y \leq z} c_R(y, \vec{x})$

Primitive recursive relations

- A relation $R \subseteq \mathbb{N}^k$ is primitive recursive if its characteristic function c_R is primitive recursive
- *iszero*

$$
\begin{aligned}
iszero(0) &= true & \qquad c_{iszero}(0) &= succ(\Pi_1^1(0)) \\
iszero(n+1) &= false & \qquad c_{iszero}(n+1) &= Z(n)
\end{aligned}
$$

- $x \leq y$ iff $iszero(x - y)$, so $c_{\leq}(x, y) = c_{iszero}(x - y)$
- $c_{\neg P} = 1 - c_P$, $c_{P \wedge Q} = c_P \cdot c_Q$
- For $Q(z, \vec{x}) = (\forall y \leq z) R(y, \vec{x})$, $c_Q(z, \vec{x}) = \prod_{y \leq z} c_R(y, \vec{x})$
- $x = y$, $x < y$, $P \vee Q$, $P \rightarrow Q$, $(\exists y \leq z) R(y, \vec{x})$ etc. obtained easily

More primitive recursion ...

- If $R(y, \vec{x})$ is a relation, $\mu y.R(y, \vec{x}) = \mu y.(1 - c_R(y, \vec{x}) = 0)$

More primitive recursion ...

- If $R(y, \vec{x})$ is a relation, $\mu y.R(y, \vec{x}) = \mu y.(1 - c_R(y, \vec{x}) = 0)$
- **Bounded μ-recursion**

$$\mu y_{\leq z}\ R(y, \vec{x}) = \begin{cases} \mu y.R(y, \vec{x}) & \text{if } (\exists y \leq z) R(y, \vec{x}) \\ z + 1 & \text{otherwise} \end{cases}$$

More primitive recursion ...

- If $R(y,\vec{x})$ is a relation, $\mu y.R(y,\vec{x}) = \mu y.(1 - c_R(y,\vec{x}) = 0)$
- **Bounded μ-recursion**

$$\mu y_{\leq z}\ R(y,\vec{x}) = \begin{cases} \mu y.R(y,\vec{x}) & \text{if } (\exists y \leq z)R(y,\vec{x}) \\ z+1 & \text{otherwise} \end{cases}$$

- Let $Q'(y,\vec{x})$ be $(\forall w \leq y)\neg R(w,\vec{x})$ and $Q(y,\vec{x})$ be $R(y,\vec{x}) \wedge Q'(y,\vec{x})$

More primitive recursion ...

- If $R(y, \vec{x})$ is a relation, $\mu y.R(y, \vec{x}) = \mu y.(1 - c_R(y, \vec{x}) = 0)$
- **Bounded μ-recursion**

$$\mu y_{\leq z}\ R(y, \vec{x}) = \begin{cases} \mu y.R(y, \vec{x}) & \text{if } (\exists y \leq z) R(y, \vec{x}) \\ z + 1 & \text{otherwise} \end{cases}$$

- Let $Q'(y, \vec{x})$ be $(\forall w \leq y)\neg R(w, \vec{x})$ and $Q(y, \vec{x})$ be $R(y, \vec{x}) \wedge Q'(y, \vec{x})$
  - If R is primitive recursive, so are Q' and Q

More primitive recursion ...

- If $R(y, \vec{x})$ is a relation, $\mu y.R(y, \vec{x}) = \mu y.(1 - c_R(y, \vec{x}) = 0)$
- **Bounded μ-recursion**

$$\mu y_{\leq z}\ R(y, \vec{x}) = \begin{cases} \mu y.R(y, \vec{x}) & \text{if } (\exists y \leq z) R(y, \vec{x}) \\ z + 1 & \text{otherwise} \end{cases}$$

- Let $Q'(y, \vec{x})$ be $(\forall w \leq y) \neg R(w, \vec{x})$ and $Q(y, \vec{x})$ be $R(y, \vec{x}) \wedge Q'(y, \vec{x})$
  - If R is primitive recursive, so are Q' and Q
- $\mu y_{\leq z}\ R(y, \vec{x}) = \sum_{y \leq z} y \cdot c_Q(y, \vec{x}) + (z + 1) \cdot c_{Q'}(y, \vec{x})$

More primitive recursion ...

- x **divides** y

$$x|y \text{ iff } (\exists z \leq y)(x \cdot z = y)$$

More primitive recursion …

- x **divides** y

$$x | y \text{ iff } (\exists z \leq y)(x \cdot z = y)$$

- x **is even**

$$even(x) \text{ iff } 2 | x$$

More primitive recursion ...

- x **divides** y

$$x|y \text{ iff } (\exists z \leq y)(x \cdot z = y)$$

- x **is even**

$$even(x) \text{ iff } 2|x$$

- x **is odd**

$$odd(x) \text{ iff } \neg even(x)$$

More primitive recursion ...

- x **divides** y

$$x|y \text{ iff } (\exists z \leq y)(x \cdot z = y)$$

- x **is even**

$$even(x) \text{ iff } 2|x$$

- x **is odd**

$$odd(x) \text{ iff } \neg even(x)$$

- x **is a prime**

$$prime(x) \text{ iff } x \geq 2 \land (\forall y \leq x)(y|x \rightarrow y = 1 \lor y = x)$$

More primitive recursion ...

- **the n-th prime**

$$\begin{aligned} Pr(0) &= 2 \\ Pr(n+1) &= \text{the smallest prime greater than } Pr(n) \\ &= \mu y_{\leq Pr(n)!+1}\,(prime(y) \wedge y > Pr(n)) \end{aligned}$$

More primitive recursion ...

- **the n-th prime**

$$
\begin{aligned}
Pr(0) &= 2 \\
Pr(n+1) &= \text{the smallest prime greater than } Pr(n) \\
&= \mu y_{\leq Pr(n)!+1}\,(prime(y) \wedge y > Pr(n))
\end{aligned}
$$

  - The (very loose) bound is guaranteed by Euclid's proof

More primitive recursion ...

- **the n-th prime**

$$
\begin{aligned}
Pr(0) &= 2 \\
Pr(n+1) &= \text{the smallest prime greater than } Pr(n) \\
&= \mu y_{\leq Pr(n)!+1}\, (prime(y) \wedge y > Pr(n))
\end{aligned}
$$

  - The (very loose) bound is guaranteed by Euclid's proof

- **the exponent of (the prime) k in the decomposition of y**

$$exp(y, k) = \mu x_{\leq y} \left[k^x | y \wedge \neg(k^{x+1} | y) \right]$$

Primitive recursive coding of the plane

- $\frac{x}{2} = \mu y_{\leq x}(2y \geq x)$

Primitive recursive coding of the plane

- $\dfrac{x}{2} = \mu y_{\leq x}(2y \geq x)$
- Primitive recursive bijection between $\mathbb{N} \times \mathbb{N}$ and $\mathbb{N}$ is given by

$$pair(x, y) = \frac{(x+y)^2 + 3x + y}{2}$$

Primitive recursive coding of the plane

- $\frac{x}{2} = \mu y_{\leq x}(2y \geq x)$
- Primitive recursive bijection between $\mathbb{N} \times \mathbb{N}$ and $\mathbb{N}$ is given by

$$pair(x, y) = \frac{(x+y)^2 + 3x + y}{2}$$

- The inverses are also primitive recursive

Primitive recursive coding of the plane

- $\dfrac{x}{2} = \mu y_{\leq x}(2y \geq x)$
- Primitive recursive bijection between $\mathbb{N} \times \mathbb{N}$ and $\mathbb{N}$ is given by

$$pair(x, y) = \frac{(x+y)^2 + 3x + y}{2}$$

- The inverses are also primitive recursive
- $fst(z) = \mu x_{\leq z}\left[(\exists y \leq z)(z = pair(x, y))\right]$

Primitive recursive coding of the plane

- $\dfrac{x}{2} = \mu y_{\leq x}(2y \geq x)$
- Primitive recursive bijection between $\mathbb{N} \times \mathbb{N}$ and $\mathbb{N}$ is given by

$$pair(x, y) = \frac{(x+y)^2 + 3x + y}{2}$$

- The inverses are also primitive recursive
- $fst(z) = \mu x_{\leq z}\left[(\exists y \leq z)(z = pair(x, y))\right]$
- $snd(z) = \mu y_{\leq z}\left[(\exists x \leq z)(z = pair(x, y))\right]$

Primitive recursive coding of sequences

- The sequence $x_1, \ldots, x_n$ (of length n) is coded by

$$Pr(0)^n \cdot Pr(1)^{x_1} \cdot Pr(2)^{x_2} \cdots Pr(n)^{x_n}$$

Primitive recursive coding of sequences

- The sequence $x_1, \ldots, x_n$ (of length n) is coded by

$$Pr(0)^n \cdot Pr(1)^{x_1} \cdot Pr(2)^{x_2} \cdots Pr(n)^{x_n}$$

- **n-th element of the sequence coded by x**

$$(x)_n = exp(x, Pr(n))$$

Primitive recursive coding of sequences

- The sequence $x_1, \ldots, x_n$ (of length n) is coded by

$$Pr(0)^n \cdot Pr(1)^{x_1} \cdot Pr(2)^{x_2} \cdots Pr(n)^{x_n}$$

- n-th element of the sequence coded by x

$$(x)_n = exp(x, Pr(n))$$

- length of sequence coded by x

$$ln(x) = (x)_0$$

Primitive recursive coding of sequences

- The sequence $x_1, \ldots, x_n$ (of length n) is coded by

$$Pr(0)^n \cdot Pr(1)^{x_1} \cdot Pr(2)^{x_2} \cdots Pr(n)^{x_n}$$

- n-th element of the sequence coded by x

$$(x)_n = exp(x, Pr(n))$$

- length of sequence coded by x

$$ln(x) = (x)_0$$

- x is a sequence number, i.e. codes a sequence

$$Seq(x) \text{ iff } (\forall n \leq x)[(n > 0 \wedge (x)_n \neq 0) \rightarrow n \leq ln(x)]$$

Turing machines

A (two-way infinite, non-deterministic) turing machine M is given by

- a finite set of states $Q = \{q_0, q_1, \ldots, q_l\}$

Turing machines

A (two-way infinite, non-deterministic) turing machine M is given by

- a finite set of states $Q = \{q_0, q_1, \ldots, q_l\}$
- q_0 is the **initial state** and q_1 is the **final state**

Turing machines

A (two-way infinite, non-deterministic) turing machine M is given by

- a finite set of states $Q = \{q_0, q_1, \ldots, q_l\}$
- q_0 is the **initial state** and q_1 is the **final state**
- The tape alphabet is $\{0, 1\}$

Turing machines

A (two-way infinite, non-deterministic) turing machine M is given by

- a finite set of states $Q = \{q_0, q_1, \ldots, q_l\}$
- q_0 is the **initial state** and q_1 is the **final state**
- The tape alphabet is $\{0, 1\}$
- a finite set of transitions of the form

$$(q_i, a) \longrightarrow (q_j, b, d)$$

where $i, j \le l$, $a, b \in \{0, 1\}$, $d \in \{L, R\}$

Turing machines

A (two-way infinite, non-deterministic) turing machine M is given by

- a finite set of states $Q = \{q_0, q_1, \ldots, q_l\}$
- q_0 is the **initial state** and q_1 is the **final state**
- The tape alphabet is $\{0, 1\}$
- a finite set of transitions of the form

$$(q_i, a) \longrightarrow (q_j, b, d)$$

  where $i, j \leq l$, $a, b \in \{0, 1\}$, $d \in \{L, R\}$
- **Meaning**: The machine, in state q_i and reading symbol a on the tape, switches to state q_j, overwriting the tape cell with the symbol b, and moves in direction specified by d (either left or right)

Turing machine: configurations

- **Initial configuration**

Turing machine: configurations

- **Initial configuration**
  - Machine is in state q_0

Turing machine: configurations

- **Initial configuration**
  - Machine is in state q_0
  - The tape only has 0's to the right of the head

Turing machine: configurations

- **Initial configuration**
  - Machine is in state q_0
  - The tape only has 0's to the right of the head
  - There are finitely many 1's to the left of the head

Turing machine: configurations

- **Initial configuration**
  - Machine is in state q_0
  - The tape only has 0's to the right of the head
  - There are finitely many 1's to the left of the head
  - The tape contents from the leftmost 1 upto the head is the input in binary

Turing machine: configurations

- **Initial configuration**
  - Machine is in state q_0
  - The tape only has 0's to the right of the head
  - There are finitely many 1's to the left of the head
  - The tape contents from the leftmost 1 upto the head is the input in binary
- **Final configuration**

Turing machine: configurations

- **Initial configuration**
  - Machine is in state q_0
  - The tape only has 0's to the right of the head
  - There are finitely many 1's to the left of the head
  - The tape contents from the leftmost 1 upto the head is the input in binary
- **Final configuration**
  - Machine is in state q_1

Turing machine: configurations

- **Initial configuration**
  - Machine is in state q_0
  - The tape only has 0's to the right of the head
  - There are finitely many 1's to the left of the head
  - The tape contents from the leftmost 1 upto the head is the input in binary
- **Final configuration**
  - Machine is in state q_1
  - The tape only has 0's to the right of the head

Turing machine: configurations

- **Initial configuration**
  - Machine is in state q_0
  - The tape only has 0's to the right of the head
  - There are finitely many 1's to the left of the head
  - The tape contents from the leftmost 1 upto the head is the input in binary
- **Final configuration**
  - Machine is in state q_1
  - The tape only has 0's to the right of the head
  - There are finitely many 1's to the left of the head

Turing machine: configurations

- **Initial configuration**
  - Machine is in state q_0
  - The tape only has 0's to the right of the head
  - There are finitely many 1's to the left of the head
  - The tape contents from the leftmost 1 upto the head is the input in binary
- **Final configuration**
  - Machine is in state q_1
  - The tape only has 0's to the right of the head
  - There are finitely many 1's to the left of the head
  - The tape contents from the leftmost 1 upto the head is the output in binary

Turing machine: configurations

- **Initial configuration**
  - Machine is in state q_0
  - The tape only has 0's to the right of the head
  - There are finitely many 1's to the left of the head
  - The tape contents from the leftmost 1 upto the head is the input in binary
- **Final configuration**
  - Machine is in state q_1
  - The tape only has 0's to the right of the head
  - There are finitely many 1's to the left of the head
  - The tape contents from the leftmost 1 upto the head is the output in binary
- **Any configuration**

Turing machine: configurations

- **Initial configuration**
  - Machine is in state q_0
  - The tape only has 0's to the right of the head
  - There are finitely many 1's to the left of the head
  - The tape contents from the leftmost 1 upto the head is the input in binary
- **Final configuration**
  - Machine is in state q_1
  - The tape only has 0's to the right of the head
  - There are finitely many 1's to the left of the head
  - The tape contents from the leftmost 1 upto the head is the output in binary
- **Any configuration**
  - Machine is in state q_i, with $0 \leq i \leq l$

Turing machine: configurations

- **Initial configuration**
  - Machine is in state q_0
  - The tape only has 0's to the right of the head
  - There are finitely many 1's to the left of the head
  - The tape contents from the leftmost 1 upto the head is the input in binary
- **Final configuration**
  - Machine is in state q_1
  - The tape only has 0's to the right of the head
  - There are finitely many 1's to the left of the head
  - The tape contents from the leftmost 1 upto the head is the output in binary
- **Any configuration**
  - Machine is in state q_i, with $0 \leq i \leq l$
  - There are only finitely many 1's on the tape

Turing machine: configurations

- **Initial configuration**
  - Machine is in state q_0
  - The tape only has 0's to the right of the head
  - There are finitely many 1's to the left of the head
  - The tape contents from the leftmost 1 upto the head is the input in binary
- **Final configuration**
  - Machine is in state q_1
  - The tape only has 0's to the right of the head
  - There are finitely many 1's to the left of the head
  - The tape contents from the leftmost 1 upto the head is the output in binary
- **Any configuration**
  - Machine is in state q_i, with $0 \leq i \leq l$
  - There are only finitely many 1's on the tape
- **Inputs and outputs are odd numbers**: $\frac{f(2m+1)-1}{2}$

Coding configurations

- A configuration is given by $pair(i, pair(x, y))$

Coding configurations

- A configuration is given by $pair(i, pair(x, y))$
  - q_i is the state

Coding configurations

- A configuration is given by $pair(i, pair(x, y))$
  - q_i is the state
  - the tape contents to the left of (and upto) the head is the binary representation of x

Coding configurations

- A configuration is given by $pair(i, pair(x, y))$
  - q_i is the state
  - the tape contents to the left of (and upto) the head is the binary representation of x
  - the **reverse** of the tape contents strictly to the right of the head is the binary representation of y

Coding configurations

- A configuration is given by $pair(i, pair(x, y))$
  - q_i is the state
  - the tape contents to the left of (and upto) the head is the binary representation of x
  - the **reverse** of the tape contents strictly to the right of the head is the binary representation of y
- **state of a configuration**: $state(n) = fst(n)$

Coding configurations

- A configuration is given by $pair(i, pair(x, y))$
  - q_i is the state
  - the tape contents to the left of (and upto) the head is the binary representation of x
  - the **reverse** of the tape contents strictly to the right of the head is the binary representation of y
- **state of a configuration**: $state(n) = fst(n)$
- **tape contents to the left**: $left(n) = fst(snd(n))$

Coding configurations

- A configuration is given by $pair(i, pair(x, y))$
  - q_i is the state
  - the tape contents to the left of (and upto) the head is the binary representation of x
  - the **reverse** of the tape contents strictly to the right of the head is the binary representation of y
- **state of a configuration**: $state(n) = fst(n)$
- **tape contents to the left**: $left(n) = fst(snd(n))$
- **tape contents to the right**: $right(n) = snd(snd(n))$

Coding configurations

- A configuration is given by $pair(i, pair(x, y))$
  - q_i is the state
  - the tape contents to the left of (and upto) the head is the binary representation of x
  - the **reverse** of the tape contents strictly to the right of the head is the binary representation of y
- **state of a configuration**: $state(n) = fst(n)$
- **tape contents to the left**: $left(n) = fst(snd(n))$
- **tape contents to the right**: $right(n) = snd(snd(n))$
- n **codes up a configuration**: $config(n) \Leftrightarrow 0 \leq state(n) \leq l$

Coding configurations

- A configuration is given by $pair(i, pair(x, y))$
  - q_i is the state
  - the tape contents to the left of (and upto) the head is the binary representation of x
  - the **reverse** of the tape contents strictly to the right of the head is the binary representation of y
- **state of a configuration**: $state(n) = fst(n)$
- **tape contents to the left**: $left(n) = fst(snd(n))$
- **tape contents to the right**: $right(n) = snd(snd(n))$
- n **codes up a configuration**: $config(n) \Leftrightarrow 0 \leq state(n) \leq l$
- n **is an initial configuration**: $initial(n) \Leftrightarrow state(n) = 0 \wedge right(n) = 0$

Coding configurations

- A configuration is given by $pair(i, pair(x, y))$
  - q_i is the state
  - the tape contents to the left of (and upto) the head is the binary representation of x
  - the reverse of the tape contents strictly to the right of the head is the binary representation of y
- **state of a configuration**: $state(n) = fst(n)$
- **tape contents to the left**: $left(n) = fst(snd(n))$
- **tape contents to the right**: $right(n) = snd(snd(n))$
- n **codes up a configuration**: $config(n) \Leftrightarrow 0 \leq state(n) \leq l$
- n **is an initial configuration**: $initial(n) \Leftrightarrow state(n) = 0 \wedge right(n) = 0$
- n **is a final configuration**: $final(n) \Leftrightarrow state(n) = 1 \wedge right(n) = 0$

Coding transitions

- Suppose t is the transition $(q_4, 0) \longrightarrow (q_8, 1, L)$

Coding transitions

- Suppose t is the transition $(q_4, 0) \longrightarrow (q_8, 1, L)$
- We define the primitive recursive predicate $step_t(c, c')$

Coding transitions

- Suppose t is the transition $(q_4, 0) \longrightarrow (q_8, 1, L)$
- We define the primitive recursive predicate $step_t(c, c')$
  - **Meaning**: t can be fired in configuration c, yielding c'

Coding transitions

- Suppose t is the transition $(q_4, 0) \longrightarrow (q_8, 1, L)$
- We define the primitive recursive predicate $step_t(c, c')$
  - **Meaning**: t can be fired in configuration c, yielding c'
- If we let $c = (i, (l, r))$ and $c' = (i', (l', r'))$

Coding transitions

- Suppose t is the transition $(q_4, 0) \longrightarrow (q_8, 1, L)$
- We define the primitive recursive predicate $step_t(c, c')$
  - **Meaning**: t can be fired in configuration c, yielding c'
- If we let $c = (i, (l, r))$ and $c' = (i', (l', r'))$
  - $i = 4$ and $i' = 8$

Coding transitions

- Suppose t is the transition $(q_4, 0) \longrightarrow (q_8, 1, L)$
- We define the primitive recursive predicate $step_t(c, c')$
  - **Meaning**: t can be fired in configuration c, yielding c'
- If we let $c = (i, (l, r))$ and $c' = (i', (l', r'))$
  - $i = 4$ and $i' = 8$
  - rightmost bit of l is 0, i.e. $even(l)$ holds

Coding transitions

- Suppose t is the transition $(q_4, 0) \longrightarrow (q_8, 1, L)$
- We define the primitive recursive predicate $step_t(c, c')$
  - **Meaning**: t can be fired in configuration c, yielding c'
- If we let $c = (i, (l, r))$ and $c' = (i', (l', r'))$
  - $i = 4$ and $i' = 8$
  - rightmost bit of l is 0, i.e. $even(l)$ holds
  - l' is got by dropping the last bit of l, .i.e. $l' = \frac{l}{2}$

Coding transitions

- Suppose t is the transition $(q_4, 0) \longrightarrow (q_8, 1, L)$
- We define the primitive recursive predicate $step_t(c, c')$
  - **Meaning**: t can be fired in configuration c, yielding c'
- If we let $c = (i, (l, r))$ and $c' = (i', (l', r'))$
  - $i = 4$ and $i' = 8$
  - rightmost bit of l is 0, i.e. $even(l)$ holds
  - l' is got by dropping the last bit of l, .i.e. $l' = \frac{l}{2}$
  - r' acquires a new rightmost bit, which is 1, i.e. $r' = 2r + 1$

Coding transitions

- Suppose t is the transition $(q_4, 0) \longrightarrow (q_8, 1, L)$
- We define the primitive recursive predicate $step_t(c, c')$
  - **Meaning**: t can be fired in configuration c, yielding c'
- If we let $c = (i, (l, r))$ and $c' = (i', (l', r'))$
  - $i = 4$ and $i' = 8$
  - rightmost bit of l is 0, i.e. $even(l)$ holds
  - l' is got by dropping the last bit of l, .i.e. $l' = \frac{l}{2}$
  - r' acquires a new rightmost bit, which is 1, i.e. $r' = 2r + 1$
- $step_t(c, c') \quad \Leftrightarrow \quad config(c) \wedge config(c') \wedge state(c) = 4 \wedge state(c') = 8 \wedge$
  $even(left(c)) \wedge 2 \cdot left(c') = left(c) \wedge$
  $right(c') = 2 \cdot right(c) + 1$

Coding transitions

- Suppose t' is the transition $(q_7, 1) \longrightarrow (q_2, 0, R)$

Coding transitions

- Suppose t' is the transition $(q_7, 1) \longrightarrow (q_2, 0, R)$
- We define the primitive recursive predicate $step_{t'}(c, c')$

Coding transitions

- Suppose t' is the transition $(q_7, 1) \longrightarrow (q_2, 0, R)$
- We define the primitive recursive predicate $step_{t'}(c, c')$
  - **Meaning**: t' can be fired in configuration c, yielding c'

Coding transitions

- Suppose t' is the transition $(q_7, 1) \longrightarrow (q_2, 0, R)$
- We define the primitive recursive predicate $step_{t'}(c, c')$
  - **Meaning**: t' can be fired in configuration c, yielding c'
- If we let $c = (i, (l, r))$ and $c' = (i', (l', r'))$

Coding transitions

- Suppose t' is the transition $(q_7, 1) \longrightarrow (q_2, 0, R)$
- We define the primitive recursive predicate $step_{t'}(c, c')$
  - **Meaning**: t' can be fired in configuration c, yielding c'
- If we let $c = (i, (l, r))$ and $c' = (i', (l', r'))$
  - $i = 7$ and $i' = 2$

Coding transitions

- Suppose t' is the transition $(q_7, 1) \longrightarrow (q_2, 0, R)$
- We define the primitive recursive predicate $step_{t'}(c, c')$
  - **Meaning**: t' can be fired in configuration c, yielding c'
- If we let $c = (i, (l, r))$ and $c' = (i', (l', r'))$
  - $i = 7$ and $i' = 2$
  - rightmost bit of l is 1, i.e. $odd(l)$ holds

Coding transitions

- Suppose t' is the transition $(q_7, 1) \longrightarrow (q_2, 0, R)$
- We define the primitive recursive predicate $step_{t'}(c, c')$
  - Meaning: t' can be fired in configuration c, yielding c'
- If we let $c = (i, (l, r))$ and $c' = (i', (l', r'))$
  - $i = 7$ and $i' = 2$
  - rightmost bit of l is 1, i.e. $odd(l)$ holds
  - Let b be the rightmost bit of r, i.e. $b = c_{odd}(r)$

Coding transitions

- Suppose t' is the transition $(q_7, 1) \longrightarrow (q_2, 0, R)$
- We define the primitive recursive predicate $step_{t'}(c, c')$
  - **Meaning**: t' can be fired in configuration c, yielding c'
- If we let $c = (i, (l, r))$ and $c' = (i', (l', r'))$
  - $i = 7$ and $i' = 2$
  - rightmost bit of l is 1, i.e. $odd(l)$ holds
  - Let b be the rightmost bit of r, i.e. $b = c_{odd}(r)$
  - l' acquires b as its rightmost bit, and second bit from the right is changed from 1 to 0, i.e. $l' = 2(l-1) + b$

Coding transitions

- Suppose t' is the transition $(q_7, 1) \longrightarrow (q_2, 0, R)$
- We define the primitive recursive predicate $step_{t'}(c, c')$
  - **Meaning**: t' can be fired in configuration c, yielding c'
- If we let $c = (i, (l, r))$ and $c' = (i', (l', r'))$
  - $i = 7$ and $i' = 2$
  - rightmost bit of l is 1, i.e. $odd(l)$ holds
  - Let b be the rightmost bit of r, i.e. $b = c_{odd}(r)$
  - l' acquires b as its rightmost bit, and second bit from the right is changed from 1 to 0, i.e. $l' = 2(l-1) + b$
  - r' is got by dropping the rightmost bit of r i.e. $r' = \frac{r}{2}$

Coding transitions

- Suppose t' is the transition $(q_7, 1) \longrightarrow (q_2, 0, R)$
- We define the primitive recursive predicate $\mathit{step}_{t'}(c, c')$
  - Meaning: t' can be fired in configuration c, yielding c'
- If we let $c = (i, (l, r))$ and $c' = (i', (l', r'))$
  - $i = 7$ and $i' = 2$
  - rightmost bit of l is 1, i.e. $\mathit{odd}(l)$ holds
  - Let b be the rightmost bit of r, i.e. $b = c_{\mathit{odd}}(r)$
  - l' acquires b as its rightmost bit, and second bit from the right is changed from 1 to 0, i.e. $l' = 2(l-1) + b$
  - r' is got by dropping the rightmost bit of r i.e. $r' = \frac{r}{2}$
- $\mathit{step}_{t'}(c, c') \quad \Leftrightarrow \quad \mathit{config}(c) \wedge \mathit{config}(c') \wedge \mathit{state}(c) = 7 \wedge \mathit{state}(c') = 2 \wedge$
  $\mathit{odd}(\mathit{left}(c)) \wedge \mathit{left}(c') = 2(\mathit{left}(c) - 1) + c_{\mathit{odd}}(\mathit{right}(c)) \wedge$
  $2 \cdot \mathit{right}(c') = \mathit{right}(c)$

Coding transitions and runs

- $step_M(c, c') \Leftrightarrow \bigvee_{t \in T} step_t(c, c')$, where T is the set of all transitions of M

Coding transitions and runs

- $step_M(c, c') \Leftrightarrow \bigvee_{t \in T} step_t(c, c')$, where T is the set of all transitions of M
- A (terminating) run of M on input m is a sequence of configurations $c_1, \ldots, c_k$

Coding transitions and runs

- $step_M(c, c') \Leftrightarrow \bigvee_{t \in T} step_t(c, c')$, where T is the set of all transitions of M
- A (terminating) run of M on input m is a sequence of configurations $c_1, \ldots, c_k$
  - c_1 is an initial configuration with $left(c_1) = m$

Coding transitions and runs

- $step_M(c, c') \Leftrightarrow \bigvee_{t \in T} step_t(c, c')$, where T is the set of all transitions of M
- A (terminating) run of M on input m is a sequence of configurations $c_1, \ldots, c_k$
  - c_1 is an initial configuration with $\mathit{left}(c_1) = m$
  - c_k is a final configuration, with the output recoverable as $\mathit{left}(c_k)$

Coding transitions and runs

- $step_M(c, c') \Leftrightarrow \bigvee_{t \in T} step_t(c, c')$, where T is the set of all transitions of M
- A (terminating) run of M on input m is a sequence of configurations $c_1, \ldots, c_k$
  - c_1 is an initial configuration with $left(c_1) = m$
  - c_k is a final configuration, with the output recoverable as $left(c_k)$
  - for all $i < k$, $step_M(c_i, c_{i+1})$ holds

Coding transitions and runs

- $step_M(c, c') \Leftrightarrow \bigvee_{t \in T} step_t(c, c')$, where T is the set of all transitions of M
- A (terminating) run of M on input m is a sequence of configurations $c_1, \ldots, c_k$
  - c_1 is an initial configuration with $left(c_1) = m$
  - c_k is a final configuration, with the output recoverable as $left(c_k)$
  - for all $i < k$, $step_M(c_i, c_{i+1})$ holds
- r **codes up a terminating run of M of length k on input m**

$$
\begin{aligned}
run_M(m, r, k) \quad \Leftrightarrow \quad & Seq(r) \wedge ln(r) = k \wedge \\
& initial((r)_1) \wedge left((r)_1) = m \wedge final((r)_k) \wedge \\
& (\forall i < k)[step_M((r)_i, (r)_{i+1})]
\end{aligned}
$$

Coding transitions and runs

- $step_M(c, c') \Leftrightarrow \bigvee_{t \in T} step_t(c, c')$, where T is the set of all transitions of M
- A (terminating) run of M on input m is a sequence of configurations $c_1, \ldots, c_k$
    - c_1 is an initial configuration with $left(c_1) = m$
    - c_k is a final configuration, with the output recoverable as $left(c_k)$
    - for all $i < k$, $step_M(c_i, c_{i+1})$ holds
- r **codes up a terminating run of M of length k on input m**

$$
\begin{aligned}
run_M(m, r, k) \quad \Leftrightarrow \quad & Seq(r) \wedge ln(r) = k \wedge \\
& initial((r)_1) \wedge left((r)_1) = m \wedge final((r)_k) \wedge \\
& (\forall i < k)[step_M((r)_i, (r)_{i+1})]
\end{aligned}
$$

- If r is a run and k is the length of r, $output = left((r)_k)$

Coding transitions and runs

- $step_M(c, c') \Leftrightarrow \bigvee_{t \in T} step_t(c, c')$, where T is the set of all transitions of M
- A (terminating) run of M on input m is a sequence of configurations $c_1, \ldots, c_k$
  - c_1 is an initial configuration with $left(c_1) = m$
  - c_k is a final configuration, with the output recoverable as $left(c_k)$
  - for all $i < k$, $step_M(c_i, c_{i+1})$ holds
- r **codes up a terminating run of M of length k on input m**

$$
\begin{aligned}
run_M(m, r, k) \quad \Leftrightarrow \quad & Seq(r) \wedge ln(r) = k \wedge \\
& initial((r)_1) \wedge left((r)_1) = m \wedge final((r)_k) \wedge \\
& (\forall i < k)[step_M((r)_i, (r)_{i+1})]
\end{aligned}
$$

- If r is a run and k is the length of r, $output = left((r)_k)$
- If $n = pair(r, k)$, $result(n) = output(fst(n), snd(n))$

Turing computable functions are recursive

- Suppose a function f is computed by a Turing machine M

Turing computable functions are recursive

- Suppose a function f is computed by a Turing machine M
- For any $m \in \mathbb{N}$, $f(m)$ can be recovered as follows

$$f(m) = result[\mu n.run_M(m, fst(n), snd(n))]$$

Turing computable functions are recursive

- Suppose a function f is computed by a Turing machine M
- For any $m \in \mathbb{N}$, $f(m)$ can be recovered as follows

$$f(m) = result[\mu n.run_M(m, fst(n), snd(n))]$$

Theorem (Kleene's normal form theorem)

Every recursive function $f : \mathbb{N}^k \to \mathbb{N}$ *can be expressed as*

$$f(\vec{n}) = h(\mu n.g(n, \vec{n}))$$

where g *and* h *are primitive recursive*

Turing computable functions are recursive

- Suppose a function f is computed by a Turing machine M
- For any $m \in \mathbb{N}$, $f(m)$ can be recovered as follows

$$f(m) = result[\mu n.run_M(m, fst(n), snd(n))]$$

Theorem (Kleene's normal form theorem)

Every recursive function $f : \mathbb{N}^k \to \mathbb{N}$ can be expressed as

$$f(\vec{n}) = h(\mu n. g(n, \vec{n}))$$

where g and h are primitive recursive

Proof.

Translate f to a Turing machine (via programs involving `for` and `while` loops), and then translate back using the above coding of runs □