Lambda calculus – encoding arithmetic

Madhavan Mukund, **S P Suresh**

Programming Language Concepts
Lecture 18, 21 March 2023

λ-calculus: syntax

- Assume a countably infinite set *Var* of variables

λ-calculus: syntax

- Assume a countably infinite set Var of variables
- The set Λ of lambda expressions is given by

$$\Lambda = x \mid \lambda x \cdot M \mid MN$$

where $x \in Var$ and $M, N \in \Lambda$.

λ-calculus: syntax

- Basic rule for computation (rewriting) is called β-**reduction** (or **contraction**)

λ-calculus: syntax

- Basic rule for computation (rewriting) is called β-**reduction** (or **contraction**)
  - $(\lambda x \cdot M)N \longrightarrow_{\beta} M[x := N]$

λ-calculus: syntax

- Basic rule for computation (rewriting) is called **β-reduction** (or **contraction**)
  - $(\lambda x \cdot M)N \longrightarrow_\beta M[x := N]$
  - $M[x := N]$: substitute **free** occurrences of x in M by N

λ-calculus: syntax

- Basic rule for computation (rewriting) is called β-**reduction** (or **contraction**)
  - $(\lambda x \cdot M)N \longrightarrow_{\beta} M[x := N]$
  - $M[x := N]$: substitute **free** occurrences of x in M by N
- We rename the bound variables in M to avoid "capturing" free variables of N in M

λ-calculus: syntax

- Basic rule for computation (rewriting) is called β-**reduction** (or **contraction**)
    - $(\lambda x \cdot M)N \longrightarrow_\beta M[x := N]$
    - $M[x := N]$: substitute **free** occurrences of x in M by N
- We rename the bound variables in M to avoid "capturing" free variables of N in M
- β-reduction can be applied in any context, replacing a subterm of the form $(\lambda x \cdot M)N$ with $M[x := N]$

λ-calculus: syntax

- Basic rule for computation (rewriting) is called β-**reduction** (or **contraction**)
  - $(\lambda x \cdot M)N \longrightarrow_{\beta} M[x := N]$
  - $M[x := N]$: substitute **free** occurrences of x in M by N
- We rename the bound variables in M to avoid "capturing" free variables of N in M
- β-reduction can be applied in any context, replacing a subterm of the form $(\lambda x \cdot M)N$ with $M[x := N]$
- Multi-step reduction is denoted $\xrightarrow{*}_{\beta}$

Encoding arithmetic

- In set theory, use nesting to encode numbers

Encoding arithmetic

- In set theory, use nesting to encode numbers
  - Encoding of n: $[n]$

Encoding arithmetic

- In set theory, use nesting to encode numbers
  - Encoding of n: $[n]$
  - $[n] = \{[0], [1], \ldots, [n-1]\}$

Encoding arithmetic

- In set theory, use nesting to encode numbers
  - Encoding of n: $[n]$
  - $[n] = \{[0], [1], \ldots, [n-1]\}$
  - Thus

Encoding arithmetic

- In set theory, use nesting to encode numbers
  - Encoding of n: $[n]$
  - $[n] = \{[0],[1],\ldots,[n-1]\}$
  - Thus
    - $[0] = \emptyset$

Encoding arithmetic

- In set theory, use nesting to encode numbers
  - Encoding of n: $[n]$
  - $[n] = \{[0],[1],\ldots,[n-1]\}$
  - Thus
    - $[0] = \varnothing$
    - $[1] = \{\varnothing\}$

Encoding arithmetic

- In set theory, use nesting to encode numbers
  - Encoding of n: $\lceil n \rceil$
  - $\lceil n \rceil = \{\lceil 0 \rceil, \lceil 1 \rceil, \ldots, \lceil n-1 \rceil\}$
  - Thus
    - $\lceil 0 \rceil = \emptyset$
    - $\lceil 1 \rceil = \{\emptyset\}$
    - $\lceil 2 \rceil = \{\emptyset, \{\emptyset\}\}$

Encoding arithmetic

- In set theory, use nesting to encode numbers
  - Encoding of n: $\lceil n \rceil$
  - $\lceil n \rceil = \{\lceil 0 \rceil, \lceil 1 \rceil, \ldots, \lceil n-1 \rceil\}$
  - Thus
    - $\lceil 0 \rceil = \emptyset$
    - $\lceil 1 \rceil = \{\emptyset\}$
    - $\lceil 2 \rceil = \{\emptyset, \{\emptyset\}\}$
    - $\lceil 3 \rceil = \{\emptyset, \{\emptyset\}, \{\emptyset, \{\emptyset\}\}\}$

Encoding arithmetic

- In set theory, use nesting to encode numbers
  - Encoding of n: $\lceil n \rceil$
  - $\lceil n \rceil = \{\lceil 0 \rceil, \lceil 1 \rceil, \ldots, \lceil n-1 \rceil\}$
  - Thus
    - $\lceil 0 \rceil = \emptyset$
    - $\lceil 1 \rceil = \{\emptyset\}$
    - $\lceil 2 \rceil = \{\emptyset, \{\emptyset\}\}$
    - $\lceil 3 \rceil = \{\emptyset, \{\emptyset\}, \{\emptyset, \{\emptyset\}\}\}$
- In λ-calculus, we encode n by the number of times we apply a function (**successor**) to an element (**zero**)

Church numerals

- $[n] = \lambda f x \cdot f^n x$

Church numerals

- $\lceil n \rceil = \lambda f x \cdot f^n x$
  - $f^0 x = x$

Church numerals

- $\lceil n \rceil = \lambda fx \cdot f^n x$
  - $f^0 x = x$
  - $f^{n+1} x = f(f^n x)$

Church numerals

- $\lceil n \rceil = \lambda fx \cdot f^n x$
  - $f^0 x = x$
  - $f^{n+1} x = f(f^n x)$
  - Thus $f^n x = f(f(\cdots(fx)\cdots))$, where f is applied repeatedly n times

Church numerals

- $[n] = \lambda f x \cdot f^n x$
  - $f^0 x = x$
  - $f^{n+1} x = f(f^n x)$
  - Thus $f^n x = f(f(\cdots(fx)\cdots))$, where f is applied repeatedly n times
- For instance

Church numerals

- $[n] = \lambda fx \cdot f^n x$
  - $f^0 x = x$
  - $f^{n+1} x = f(f^n x)$
  - Thus $f^n x = f(f(\cdots(fx)\cdots))$, where f is applied repeatedly n times
- For instance
  - $[0] = \lambda fx \cdot x$

Church numerals

- $\lceil n \rceil = \lambda f x \cdot f^n x$
  - $f^0 x = x$
  - $f^{n+1} x = f(f^n x)$
  - Thus $f^n x = f(f(\cdots(fx)\cdots))$, where f is applied repeatedly n times
- For instance
  - $\lceil 0 \rceil = \lambda f x \cdot x$
  - $\lceil 1 \rceil = \lambda f x \cdot f x$

Church numerals

- $\lceil n \rceil = \lambda f x \cdot f^n x$
  - $f^0 x = x$
  - $f^{n+1} x = f(f^n x)$
  - Thus $f^n x = f(f(\cdots(fx)\cdots))$, where f is applied repeatedly n times
- For instance
  - $\lceil 0 \rceil = \lambda f x \cdot x$
  - $\lceil 1 \rceil = \lambda f x \cdot f x$
  - $\lceil 2 \rceil = \lambda f x \cdot f(f x)$

Church numerals

- $\lceil n \rceil = \lambda fx \cdot f^n x$
  - $f^0 x = x$
  - $f^{n+1} x = f(f^n x)$
  - Thus $f^n x = f(f(\cdots(fx)\cdots))$, where f is applied repeatedly n times
- For instance
  - $\lceil 0 \rceil = \lambda fx \cdot x$
  - $\lceil 1 \rceil = \lambda fx \cdot fx$
  - $\lceil 2 \rceil = \lambda fx \cdot f(fx)$
  - $\lceil 3 \rceil = \lambda fx \cdot f(f(fx))$

Church numerals

- $[n] = \lambda fx \cdot f^n x$
  - $f^0 x = x$
  - $f^{n+1} x = f(f^n x)$
  - Thus $f^n x = f(f(\cdots(fx)\cdots))$, where f is applied repeatedly n times
- For instance
  - $[0] = \lambda fx \cdot x$
  - $[1] = \lambda fx \cdot fx$
  - $[2] = \lambda fx \cdot f(fx)$
  - $[3] = \lambda fx \cdot f(f(fx))$
  - $\ldots$

Church numerals

- $[n] = \lambda fx \cdot f^n x$
  - $f^0 x = x$
  - $f^{n+1} x = f(f^n x)$
  - Thus $f^n x = f(f(\cdots(fx)\cdots))$, where f is applied repeatedly n times
- For instance
  - $[0] = \lambda fx \cdot x$
  - $[1] = \lambda fx \cdot fx$
  - $[2] = \lambda fx \cdot f(fx)$
  - $[3] = \lambda fx \cdot f(f(fx))$
  - $\ldots$
- $[n]gy = (\lambda fx \cdot f(\cdots(fx)\cdots))gy \xrightarrow{*}_\beta g(\cdots(gy)\cdots) = g^n y$

Encoding arithmetic functions

- **Successor function**: $succ(n) = n + 1$

Encoding arithmetic functions

- **Successor function**: $succ(n) = n + 1$
- $[succ] = \lambda pfx \cdot f(pfx)$

Encoding arithmetic functions

- **Successor function**: $succ(n) = n + 1$
- $\left[succ\right] = \lambda pfx \cdot f(pfx)$
- For all n, $\left[succ\right]\left[n\right] \xrightarrow{*}_{\beta} \left[n+1\right]$

Encoding arithmetic functions

- **Successor function**: $succ(n) = n + 1$
- $[succ] = \lambda pfx \cdot f(pfx)$
- For all n, $[succ][n] \xrightarrow{*}_{\beta} [n+1]$
  - $[succ][n]$

$$
\begin{array}{lll}
(\lambda pfx \cdot f(pfx))[n] & \longrightarrow_{\beta} & \lambda fx \cdot f([n]fx) \\
 & \xrightarrow{*}_{\beta} & \lambda fx \cdot f(f^n x) \\
 & = & \lambda fx \cdot f^{n+1} x \\
 & = & [n+1]
\end{array}
$$

Encoding arithmetic functions

- **Addition**: $plus(m,n) = m + n$

Encoding arithmetic functions

- **Addition**: $plus(m,n) = m + n$
- $\left[plus\right] = \lambda pqfx \cdot pf(qfx)$

Encoding arithmetic functions

- **Addition**: $plus(m,n) = m + n$
- $[plus] = \lambda pqfx \cdot pf(qfx)$
- For all m and n, $[plus][m][n] \xrightarrow{*}_{\beta} [m+n]$

Encoding arithmetic functions

- **Addition**: $plus(m,n) = m+n$
- $\lceil plus \rceil = \lambda pqfx \cdot pf(qfx)$
- For all m and n, $\lceil plus \rceil \lceil m \rceil \lceil n \rceil \xrightarrow{*}_{\beta} \lceil m+n \rceil$
  - $\lceil plus \rceil \lceil m \rceil \lceil n \rceil$

$$
\begin{array}{lll}
(\lambda pqfx \cdot pf(qfx)) \lceil m \rceil \lceil n \rceil & \longrightarrow_{\beta} & (\lambda qfx \cdot \lceil m \rceil f(qfx)) \lceil n \rceil \\
& \longrightarrow_{\beta} & \lambda fx \cdot \lceil m \rceil f(\lceil n \rceil fx) \\
& \xrightarrow{*}_{\beta} & \lambda fx \cdot f^m(\lceil n \rceil fx) \\
& \xrightarrow{*}_{\beta} & \lambda fx \cdot f^m(f^n x) \\
& = & \lambda fx \cdot f^{m+n} x \\
& = & \lceil m+n \rceil
\end{array}
$$

Encoding arithmetic functions

- **Multiplication**: $mult(m, n) = mn$

Encoding arithmetic functions

- **Multiplication**: $mult(m,n) = mn$
- $[mult] = \lambda pqf \cdot p(qf)$

Encoding arithmetic functions

- **Multiplication**: $mult(m,n) = mn$
- $\lceil mult \rceil = \lambda pqf \cdot p(qf)$
- For all $m \geq 0$, $(\lceil n \rceil f)^m y \xrightarrow{*}_\beta f^{mn} y$

Encoding arithmetic functions

- **Multiplication**: $mult(m,n) = mn$
- $[\![mult]\!] = \lambda pqf \cdot p(qf)$
- For all $m \geq 0$, $([\![n]\!]f)^m y \xrightarrow{*}_\beta f^{mn} y$
    - $([\![n]\!]f)^0 y = y = f^{0 \cdot n} y$

Encoding arithmetic functions

- **Multiplication**: $mult(m,n) = mn$
- $\lceil mult \rceil = \lambda pqf \cdot p(qf)$
- For all $m \geq 0$, $(\lceil n \rceil f)^m y \xrightarrow{*}_\beta f^{mn} y$
  - $(\lceil n \rceil f)^0 y = y = f^{0 \cdot n} y$
  - $$\begin{aligned} (\lceil n \rceil f)^{m+1} y &= (\lceil n \rceil f)((\lceil n \rceil f)^m y) \\ &\xrightarrow{*}_\beta \lceil n \rceil f(f^{mn} y) \\ &\xrightarrow{*}_\beta f^n(f^{mn} y) = f^{mn+n} y = f^{(m+1)n} y \end{aligned}$$

Encoding arithmetic functions

- **Multiplication**: $mult(m,n) = mn$
- $[mult] = \lambda pqf \cdot p(qf)$
- For all $m \geq 0$, $([n]f)^m y \xrightarrow{*}_\beta f^{mn} y$
  - $([n]f)^0 y = y = f^{0 \cdot n} y$
  - $$\begin{aligned} ([n]f)^{m+1} y &= ([n]f)(([n]f)^m y) \\ &\xrightarrow{*}_\beta [n]f(f^{mn} y) \\ &\xrightarrow{*}_\beta f^n(f^{mn} y) = f^{mn+n} y = f^{(m+1)n} y \end{aligned}$$
- For all m and n, $[mult][m][n] \xrightarrow{*}_\beta [mn]$

Encoding arithmetic functions

- **Multiplication**: $mult(m,n) = mn$
- $[mult] = \lambda pqf \cdot p(qf)$
- For all $m \geq 0$, $([n]f)^m y \xrightarrow{*}_\beta f^{mn} y$
  - $([n]f)^0 y = y = f^{0 \cdot n} y$
  - $$\begin{aligned} ([n]f)^{m+1} y &= ([n]f)(([n]f)^m y) \\ &\xrightarrow{*}_\beta [n]f(f^{mn} y) \\ &\xrightarrow{*}_\beta f^n(f^{mn} y) = f^{mn+n} y = f^{(m+1)n} y \end{aligned}$$
- For all m and n, $[mult][m][n] \xrightarrow{*}_\beta [mn]$
  - $$\begin{aligned} (\lambda pqf \cdot p(qf))[m][n] &\xrightarrow{*}_\beta \lambda f \cdot [m]([n]f) \\ &= \lambda f \cdot (\lambda gy \cdot g^m y)([n]f) \\ &\xrightarrow{*}_\beta \lambda f \cdot (\lambda y \cdot ([n]f)^m y) \\ &\xrightarrow{*}_\beta \lambda fy \cdot f^{mn} y = [mn] \end{aligned}$$

Encoding arithmetic functions

- **Exponentiation**: $exp(m, n) = n^m$

Encoding arithmetic functions

- **Exponentiation**: $exp(m, n) = n^m$
- $[exp] = \lambda pq.pq$

Encoding arithmetic functions

- **Exponentiation**: $exp(m, n) = n^m$
- $[exp] = \lambda pq.pq$
- For all $m \geq 1$ and $n \geq 0$, $[exp][m][n] \xrightarrow{*}_{\beta} [n^m]$

Encoding arithmetic functions

- **Exponentiation**: $exp(m, n) = n^m$
- $[exp] = \lambda pq.pq$
- For all $m \geq 1$ and $n \geq 0$, $[exp][m][n] \xrightarrow{*}_\beta [n^m]$
  - **Proof**: Exercise!

Computability

- Church numerals encode $n \in \mathbb{N}$

Computability

- Church numerals encode $n \in \mathbb{N}$
- Can we encode computable functions $f : \mathbb{N}^k \to \mathbb{N}$?

Computability

- Church numerals encode $n \in \mathbb{N}$
- Can we encode computable functions $f : \mathbb{N}^k \to \mathbb{N}$?
  - Let $[f]$ be the encoding of f

Computability

- Church numerals encode $n \in \mathbb{N}$
- Can we encode computable functions $f : \mathbb{N}^k \to \mathbb{N}$?
  - Let $[f]$ be the encoding of f
  - We want $[f][n_1]\cdots[n_k] \xrightarrow{*}_\beta [f(n_1,\ldots,n_k)]$

Computability

- Church numerals encode $n \in \mathbb{N}$
- Can we encode computable functions $f : \mathbb{N}^k \to \mathbb{N}$?
  - Let $[f]$ be the encoding of f
  - We want $[f][n_1]\cdots[n_k] \xrightarrow{*}_\beta [f(n_1,\ldots,n_k)]$
- We need a syntax for computable functions

Recursive functions

- Recursive functions [**Dedekind**, **Skolem**, **Gödel**, **Kleene**]

Recursive functions

- Recursive functions [**Dedekind**, **Skolem**, **Gödel**, **Kleene**]
  - Equivalent to Turing machines

Recursive functions

- Recursive functions [**Dedekind**, **Skolem**, **Gödel**, **Kleene**]
  - Equivalent to Turing machines
- $f : \mathbb{N}^k \rightarrow \mathbb{N}$ is obtained by **composition** from $g : \mathbb{N}^l \rightarrow \mathbb{N}$ and $h_1, \ldots, h_l : \mathbb{N}^k \rightarrow \mathbb{N}$ if

$$f(\vec{n}) = g(h_1(\vec{n}), \ldots, h_l(\vec{n}))$$

Recursive functions

- Recursive functions [**Dedekind**, **Skolem**, **Gödel**, **Kleene**]
  - Equivalent to Turing machines
- $f : \mathbb{N}^k \to \mathbb{N}$ is obtained by **composition** from $g : \mathbb{N}^l \to \mathbb{N}$ and $h_1, \ldots, h_l : \mathbb{N}^k \to \mathbb{N}$ if

$$f(\vec{n}) = g(h_1(\vec{n}), \ldots, h_l(\vec{n}))$$

- **Notation**: $f = g \circ (h_1, h_2, \ldots, h_l)$

Recursive functions

- $f : \mathbb{N}^{k+1} \to \mathbb{N}$ is obtained by **primitive recursion** from $g : \mathbb{N}^k \to \mathbb{N}$ and $h : \mathbb{N}^{k+2} \to \mathbb{N}$ if

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

Recursive functions

- $f : \mathbb{N}^{k+1} \longrightarrow \mathbb{N}$ is obtained by **primitive recursion** from $g : \mathbb{N}^k \longrightarrow \mathbb{N}$ and $h : \mathbb{N}^{k+2} \longrightarrow \mathbb{N}$ if

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

- **Note** If g and h are total functions, so is f

Recursive functions

- $f : \mathbb{N}^{k+1} \longrightarrow \mathbb{N}$ is obtained by **primitive recursion** from $g : \mathbb{N}^k \longrightarrow \mathbb{N}$ and $h : \mathbb{N}^{k+2} \longrightarrow \mathbb{N}$ if

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

Recursive functions

- $f : \mathbb{N}^{k+1} \longrightarrow \mathbb{N}$ is obtained by **primitive recursion** from $g : \mathbb{N}^k \longrightarrow \mathbb{N}$ and $h : \mathbb{N}^{k+2} \longrightarrow \mathbb{N}$ if

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

- Equivalent to 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

- $f : \mathbb{N}^k \to \mathbb{N}$ is obtained by μ-**recursion** or **minimization** from $g : \mathbb{N}^{k+1} \to \mathbb{N}$ if

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

Recursive functions

- $f : \mathbb{N}^k \to \mathbb{N}$ is obtained by μ**-recursion** or **minimization** from $g : \mathbb{N}^{k+1} \to \mathbb{N}$ if

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

- **Notation**: $f(\vec{n}) = \mu i\,(g(i,\vec{n}) = 0)$

Recursive functions

- $f : \mathbb{N}^k \to \mathbb{N}$ is obtained by μ**-recursion** or **minimization** from $g : \mathbb{N}^{k+1} \to \mathbb{N}$ if

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

- **Notation**: $f(\vec{n}) = \mu i\,(g(i,\vec{n}) = 0)$
- f need not be total even if g is

Recursive functions

- $f : \mathbb{N}^k \rightarrow \mathbb{N}$ is obtained by **μ-recursion** or **minimization** from $g : \mathbb{N}^{k+1} \rightarrow \mathbb{N}$ if

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

- **Notation**: $f(\vec{n}) = \mu i\,(g(i,\vec{n}) = 0)$
- f need not be total even if g is
- If $f(\vec{n}) = i$, then $g(j,\vec{n})$ is defined for all $j \leqslant i$

Recursive functions

- $f : \mathbb{N}^k \to \mathbb{N}$ is obtained by μ-**recursion** or **minimization** from $g : \mathbb{N}^{k+1} \to \mathbb{N}$ if

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

Recursive functions

- $f : \mathbb{N}^k \to \mathbb{N}$ is obtained by **μ-recursion** or **minimization** from $g : \mathbb{N}^{k+1} \to \mathbb{N}$ if

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

- Equivalent to a `while` loop:

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

Recursive functions

- The class of **primitive recursive functions** is the smallest class of functions

Recursive functions

- The class of **primitive recursive functions** is the smallest class of functions
  1. containing the **initial functions**

Recursive functions

- The class of **primitive recursive functions** is the smallest class of functions
  1. containing the **initial functions**

     Zero $Z(n) = 0$

Recursive functions

- The class of **primitive recursive functions** is the smallest class of functions
  1. containing the **initial functions**

     Zero $Z(n) = 0$

     Successor $S(n) = n + 1$

Recursive functions

- The class of **primitive recursive functions** is the smallest class of functions
  1. containing the **initial functions**

Zero	$Z(n) = 0$
Successor	$S(n) = n + 1$
Projection	$\Pi_i^k(n_1, \ldots, n_k) = n_i$

Recursive functions

- The class of **primitive recursive functions** is the smallest class of functions
  1. containing the **initial functions**

Zero	$Z(n) = 0$
Successor	$S(n) = n + 1$
Projection	$\Pi_i^k(n_1, \ldots, n_k) = n_i$

  2. closed under composition and primitive recursion

Recursive functions

- The class of **primitive recursive functions** is the smallest class of functions
  1. containing the **initial functions**

     Zero $Z(n) = 0$

     Successor $S(n) = n + 1$

     Projection $\Pi_i^k(n_1, \ldots, n_k) = n_i$
  2. closed under composition and primitive recursion
- The class of **(partial) recursive functions** is the smallest class of functions

Recursive functions

- The class of **primitive recursive functions** is the smallest class of functions
  1. containing the **initial functions**

     Zero $Z(n) = 0$

     Successor $S(n) = n + 1$

     Projection $\Pi_i^k(n_1, \ldots, n_k) = n_i$
  2. closed under composition and primitive recursion
- The class of **(partial) recursive functions** is the smallest class of functions
  1. containing the initial functions

Recursive functions

- The class of **primitive recursive functions** is the smallest class of functions
  1. containing the **initial functions**

Zero	$Z(n) = 0$
Successor	$S(n) = n + 1$
Projection	$\Pi_i^k(n_1, \ldots, n_k) = n_i$

  2. closed under composition and primitive recursion
- The class of **(partial) recursive functions** is the smallest class of functions
  1. containing the initial functions
  2. closed under composition, primitive recursion and minimization

Recursive functions: Examples

- $f(n) = n + 2$ is $S \circ S$

Recursive functions: Examples

- $f(n) = n + 2$ is $S \circ S$
- $plus(n, m) = n + m$ is got by primitive recursion from $g = \Pi_1^1$ and $h = S \circ \Pi_2^3$

$$
\begin{array}{lllll}
plus(0, m) & = & g(m) & = & \Pi_1^1(m) \\
 & & & = & m \\
plus(n+1, m) & = & h(n, plus(n, m), m) & & \\
 & = & (S \circ \Pi_2^3)(n, plus(n, m), m) & = & S(plus(n, m)) \\
 & & & = & (n + m) + 1 \\
 & & & = & (n + 1) + m
\end{array}
$$

Recursive functions: Examples

- $mult(n,m) = nm$ is got by primitive recursion from $g = Z$ and $h = plus \circ (\Pi^3_2, \Pi^3_3)$

$$
\begin{aligned}
mult(0,m) &= g(m) && = Z(m) \\
& && = 0 \\
mult(n+1,m) &= h(n, mult(n,m), m) \\
&= (plus \circ (\Pi^3_2, \Pi^3_3))(n, mult(n,m), m) \\
& && = nm + m \\
& && = (n+1)m
\end{aligned}
$$

Recursive functions: Examples

- $exp(n,m) = m^n$ is got by primitive recursion from $g = S \circ Z$ and $h = mult \circ (\Pi^3_2, \Pi^3_3)$

$$
\begin{aligned}
exp(0,m) &= g(m) &&= (S \circ Z)(m) \\
& &&= 1 \\
exp(n+1,m) &= h(n, exp(n,m), m) \\
&= (mult \circ (\Pi^3_2, \Pi^3_3))(n, exp(n,m), m) \\
& &&= m^n \cdot m \\
& &&= m^{n+1}
\end{aligned}
$$

Recursive functions: Examples

- Define $pred(n) = \begin{cases} 0 & \text{if } n = 0 \\ n - 1 & \text{otherwise} \end{cases}$

Recursive functions: Examples

- Define $pred(n) = \begin{cases} 0 & \text{if } n = 0 \\ n-1 & \text{otherwise} \end{cases}$
- $pred(n) = f(n,n)$ where f is got by primitive recursion from $g = Z$ and $h = \Pi^3_1$

$$\begin{array}{llllll} f(0,m) & = & g(m) & = & Z(m) & = & 0 \\ f(n+1,m) & = & h(n,f(n,m),m) & = & \Pi^3_1(n,f(n,m),m) & = & n \end{array}$$

$$\begin{array}{llll} pred(0) & = & f(0,0) & = & 0 \\ pred(n+1) & = & f(n+1,n+1) & = & n \end{array}$$

Recursive functions: Examples

- Define $m \dot{-} n = \begin{cases} 0 & \text{if } m \leqslant n \\ m - n & \text{otherwise} \end{cases}$

Recursive functions: Examples

- Define $m \dot{-} n = \begin{cases} 0 & \text{if } m \leqslant n \\ m - n & \text{otherwise} \end{cases}$
- $m \dot{-} n = f(n, m)$ where f is got by primitive recursion from $g = \Pi_1^1$ and $h = pred \circ \Pi_2^3$

$$
\begin{aligned}
f(0, m) &= g(m) &&= \Pi_1^1(m) \\
& &&= m &&= m \dot{-} 0 \\
f(n+1, m) &= h(n, f(n, m), m) &&= pred(\Pi_2^3(n, f(n, m), m)) \\
& &&= pred(m \dot{-} n) &&= m \dot{-} (n+1)
\end{aligned}
$$

Recursive functions: Examples

- Define $m \dot{-} n = \begin{cases} 0 & \text{if } m \leqslant n \\ m - n & \text{otherwise} \end{cases}$
- $m \dot{-} n = f(n, m)$ where f is got by primitive recursion from $g = \Pi_1^1$ and $h = pred \circ \Pi_2^3$

$$
\begin{aligned}
f(0, m) &= g(m) && = \Pi_1^1(m) \\
& && = m && = m \dot{-} 0 \\
f(n+1, m) &= h(n, f(n, m), m) && = pred(\Pi_2^3(n, f(n, m), m)) \\
& && = pred(m \dot{-} n) && = m \dot{-} (n+1)
\end{aligned}
$$

- **Note the recursion on the second argument!**

Recursive functions: Examples

- Define $m \dot{-} n = \begin{cases} 0 & \text{if } m \leqslant n \\ m - n & \text{otherwise} \end{cases}$
- $m \dot{-} n = f(n, m)$ where f is got by primitive recursion from $g = \Pi_1^1$ and $h = pred \circ \Pi_2^3$

$$
\begin{array}{lcllcl}
f(0,m) & = & g(m) & = & \Pi_1^1(m) & \\
 & & & = & m & = \; m \dot{-} 0 \\
f(n+1,m) & = & h(n, f(n,m), m) & = & pred(\Pi_2^3(n, f(n,m), m)) & \\
 & & & = & pred(m \dot{-} n) & = \; m \dot{-} (n+1)
\end{array}
$$

- **Note the recursion on the second argument!**
- $f(m) = \log_2 m$ is defined by minimization from $g(n, m) = m \dot{-} 2^n$

Recursive functions: Examples

- Define $m \dot{-} n = \begin{cases} 0 & \text{if } m \leqslant n \\ m - n & \text{otherwise} \end{cases}$
- $m \dot{-} n = f(n,m)$ where f is got by primitive recursion from $g = \Pi_1^1$ and $h = pred \circ \Pi_2^3$

$$
\begin{aligned}
f(0,m) &= g(m) &&= \Pi_1^1(m) \\
& &&= m &&= m \dot{-} 0 \\
f(n+1,m) &= h(n,f(n,m),m) &&= pred(\Pi_2^3(n,f(n,m),m)) \\
& &&= pred(m \dot{-} n) &&= m \dot{-} (n+1)
\end{aligned}
$$

- **Note the recursion on the second argument!**
- $f(m) = \log_2 m$ is defined by minimization from $g(n,m) = m \dot{-} 2^n$
  - First n such that $m \leqslant 2^n$ is $\lceil \log_2 m \rceil$