Programming Language Concepts: Lecture 16

S P Suresh

March 10, 2021

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

$$\Lambda = x \mid \lambda x.M \mid M N$$

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

λ-calculus: syntax

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

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

  where $x \in Var$ and $M, N \in \Lambda$.
- Basic rule for computation (rewriting) is called **β-reduction** (or **contraction**)

λ-calculus: syntax

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

$$\Lambda = x \mid \lambda x.M \mid M\,N$$

  where $x \in Var$ and $M, N \in \Lambda$.
- Basic rule for computation (rewriting) is called **β-reduction** (or **contraction**)
  - $(\lambda x.M)N \longrightarrow_{\beta} M[x := N]$

λ-calculus: syntax

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

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

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

λ-calculus: syntax

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

  $$\Lambda = x \mid \lambda x.M \mid M\,N$$

  where $x \in Var$ and $M, N \in \Lambda$.
- Basic rule for computation (rewriting) is called **β-reduction** (or **contraction**)
  - $(\lambda x.M)N \longrightarrow_\beta M[x := N]$
  - $M[x := N]$: 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

Church numerals

- $[n] = \lambda f\, x.\, f^n x$

Church numerals

- $[n] = \lambda f\,x.f^n x$
    - $f^0 x = x$

Church numerals

- $[n] = \lambda f x . f^n x$
  - $f^0 x = x$
  - $f^{n+1} x = f(f^n x)$

Church numerals

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

Church numerals

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

Church numerals

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

Church numerals

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

Church numerals

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

Church numerals

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

Church numerals

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

Church numerals

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

Encoding arithmetic functions

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

Encoding arithmetic functions

- **Successor function**: $succ(n) = n + 1$
- $[succ] = \lambda p\, f\, x. f\, (p\, f\, x)$

Encoding arithmetic functions

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

Encoding arithmetic functions

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

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

Encoding arithmetic functions

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

Encoding arithmetic functions

- **Addition**: $plus(m, n) = m + n$
- $[plus] = \lambda p\, q\, f\, x.\, p\, f\, (q\, f\, x)$

Encoding arithmetic functions

- **Addition**: $plus(m, n) = m + n$
- $[plus] = \lambda p q f x . p f (q f x)$
- For all m and n, $[plus][m+n] \xrightarrow{*}_{\beta} [m+n]$

Encoding arithmetic functions

- **Addition**: $plus(m,n) = m + n$
- $[plus] = \lambda p q f x . p f (q f x)$
- For all m and n, $[plus][m+n] \xrightarrow{*}_{\beta} [m+n]$
  - $[plus][m][n]$

$$
\begin{array}{lll}
(\lambda p q f x . p f (q f x))[m][n] & \longrightarrow_{\beta} & (\lambda q f x . [m] f (q f x))[n] \\
 & \longrightarrow_{\beta} & \lambda f x . [m] f ([n] f x) \\
 & \xrightarrow{*}_{\beta} & \lambda f x . f^{m} ([n] f x) \\
 & \xrightarrow{*}_{\beta} & \lambda f x . f^{m} (f^{n} x) \\
 & = & \lambda f x . f^{m+n} x \\
 & = & [m+n]
\end{array}
$$

Encoding arithmetic functions

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

Encoding arithmetic functions

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

Encoding arithmetic functions

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

Encoding arithmetic functions

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

Encoding arithmetic functions

- **Multiplication**: $mult(m,n) = mn$
- $[mult] = \lambda pqf.p(qf)$
- 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**: $mult(m, n) = mn$
- $[mult] = \lambda pqf.p(qf)$
- 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, $[mult][m][n] \xrightarrow{*}_\beta [mn]$
  - $$\begin{aligned} (\lambda pqf.p(qf))[m][n] &\xrightarrow{*}_\beta \lambda f.[m]([n]f) \\ &= \lambda f.(\lambda g y.g^m y)([n]f) \\ &\xrightarrow{*}_\beta \lambda f.(\lambda y.([n]f)^m y) \\ &\xrightarrow{*}_\beta \lambda f.\lambda y.f^{mn} y = [mn] \end{aligned}$$

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
- $f : \mathbb{N}^k \to \mathbb{N}$ 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

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

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

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

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

Recursive functions

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

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

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

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

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

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

- 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

- $f : \mathbb{N}^k \to \mathbb{N}$ 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
- If $f(\vec{n}) = i$, then $g(j, \vec{n})$ is defined for all $j \leq i$

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**

$$\begin{aligned} \text{Zero} \quad & Z(n) = 0 \\ \text{Successor} \quad & S(n) = n + 1 \end{aligned}$$

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**

     $$\begin{aligned} \text{Zero} \quad & Z(n) = 0 \\ \text{Successor} \quad & S(n) = n + 1 \\ \text{Projection} \quad & \Pi_i^k(n_1, \ldots, n_k) = n_i \end{aligned}$$

  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 $h = S \circ \Pi^3_2$

$$
\begin{array}{lclcl}
plus(0, m) & = & g(m) & = & \Pi^1_1(m) \\
 & & & = & m \\
plus(n+1, m) & = & h(n, plus(n, m), m) & & \\
 & = & (S \circ \Pi^3_2)(n, plus(n, m), m) & = & S(plus(n, m)) \\
 & & & = & (n + m) + 1 \\
 & & & = & (n + 1) + m
\end{array}
$$

Recursive functions: Examples

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

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

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

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

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

- $f(m) = \log_2 m$ is defined by minimization from $g(n, m) = m - 2^n$

Recursive functions: Examples

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

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

- $f(m) = \log_2 m$ is defined by minimization from $g(n, m) = m - 2^n$
  - First n such that $m - 2^n = 0$ is $\lceil \log_2 m \rceil$

Recursive functions: Examples

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

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

- $f(m) = \log_2 m$ is defined by minimization from $g(n, m) = m - 2^n$
  - First n such that $m - 2^n = 0$ is $\lceil \log_2 m \rceil$
  - $p - q$ is 0 whenever $p \leqslant q$

Recursive functions: Examples

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

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

- $f(m) = \log_2 m$ is defined by minimization from $g(n, m) = m - 2^n$
  - First n such that $m - 2^n = 0$ is $\lceil \log_2 m \rceil$
  - $p - q$ is 0 whenever $p \leqslant q$
  - We will see a definiition of subtraction later