Programming Language Concepts: Lecture 14

S P Suresh

Chennai Mathematical Institute

spsuresh@cmi.ac.in

http://www.cmi.ac.in/~spsuresh/teaching/plc16

March 16, 2016

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

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

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

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

  where $x \in \mathit{Var}$ 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

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

  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
- 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: $plus(m,n) = m+n$
- $[plus] = \lambda pqfx.pf(qfx)$

Encoding arithmetic functions

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

Encoding arithmetic functions

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

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

Encoding arithmetic functions

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

Encoding arithmetic functions

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

Encoding arithmetic functions

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

Encoding arithmetic functions

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

Encoding arithmetic functions

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

Encoding arithmetic functions

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

Encoding arithmetic functions

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

Encoding arithmetic functions

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

Encoding arithmetic functions

- Exponentiation: $exp(m, n) = m^n$
  - $exp(0, 0)$ is taken to be 1

Encoding arithmetic functions

- Exponentiation: $exp(m,n) = m^n$
  - $exp(0,0)$ is taken to be 1
- $[exp] = \lambda pqfx.qpfx$

Encoding arithmetic functions

- Exponentiation: $exp(m,n) = m^n$
  - $exp(0,0)$ is taken to be 1
- $[exp] = \lambda pqfx.qpfx$
- For all m and n, $[exp][m][n] \xrightarrow{*}_{\beta} [m^n]$

Encoding arithmetic functions

- Exponentiation: $exp(m,n) = m^n$
  - $exp(0,0)$ is taken to be 1
- $[exp] = \lambda pqfx.qpfx$
- For all m and n, $[exp][m][n] \xrightarrow{*}_\beta [m^n]$
  - 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 [Dedekind, Skolem, Gödel, Kleene]
    - Equivalent to Turing machines

Definition
$f : \mathbb{N}^k \to \mathbb{N}$ is obtained by composition from $g : \mathbb{N}^\ell \to \mathbb{N}$ and $h_1, \ldots, h_\ell : \mathbb{N}^k \to \mathbb{N}$ if

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

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

Definition

$f : \mathbb{N}^k \to \mathbb{N}$ is obtained by composition from $g : \mathbb{N}^\ell \to \mathbb{N}$ and $h_1, \ldots, h_\ell : \mathbb{N}^k \to \mathbb{N}$ if

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

- Notation: $f = g \circ (h_1, h_2, \ldots, h_\ell)$

Recursive functions

Definition

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

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

Definition

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

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

- 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;
```

Definition

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

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

- 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;
```

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

Definition

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

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

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

Definition

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

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

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

- Equivalent to computing a `while` loop:

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

Definition

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

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

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

- Equivalent to computing a `while` loop:

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

- f need not be total even if g is

Definition

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

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

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

- Equivalent to computing a `while` loop:

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

- f need not be total even if g is
- If $f(\vec{n}) = n$, then $g(m,\vec{n})$ is defined for all $m \leq n$

Definition

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

Recursive functions

Definition

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

1. containing the initial functions

Definition

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

1. containing the initial functions

   Zero $Z(n) = 0$

Definition

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

1. containing the initial functions

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

Definition

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	$\Pi_i^k(n_1, \ldots, n_k) = n_i$

Definition

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

Definition

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

Definition

The class of (partial) recursive functions is the smallest class of functions

Recursive functions

Definition

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

1. containing the initial functions

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

2. closed under composition and primitive recursion

Definition

The class of (partial) recursive functions is the smallest class of functions

1. containing the initial functions

Recursive functions

Definition

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

Definition

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

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

Recursive functions: Examples

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

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

Recursive functions: Examples

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

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

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

- $mult(n,m) = nm$ is got by primitive recursion from $g = Z$ and $h = plus \circ (\Pi_2^3, \Pi_3^3)$

$$\begin{array}{lcll} mult(0,m) & = & g(m) & = Z(m) \\ & & & = 0 \\ mult(n+1,m) & = & h(n, plus(n,m), m) & \\ & = & plus \circ (\Pi_2^3, \Pi_3^3)(n, mult(n,m), m) & \\ & & & = nm + m \\ & & & = (n+1)m \end{array}$$

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

Recursive functions: Examples

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

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

- $f(m) = \log_2 m$ is defined by minimization from $g(n,m) = m - 2^n$
  - First n such that $m - 2^n = 0$ is $\log_2 m$
  - Not defined for all m!