Programming Language Concepts: Lecture 15

S P Suresh

Chennai Mathematical Institute
spsuresh@cmi.ac.in
http://www.cmi.ac.in/~spsuresh/teaching/plc16

March 28, 2016

Encoding arithmetic functions

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

Encoding arithmetic functions

- $[n] = \lambda f x. f^n x$
  - $f^n x = f(f(\cdots(f x)\cdots))$, where f is applied repeatedly n times

Encoding arithmetic functions

- $[n] = \lambda f x . f^n x$
  - $f^n x = f(f(\cdots(f x)\cdots))$, where f is applied repeatedly n times
- Successor: $[succ] = \lambda p f x . f(p f x)$

Encoding arithmetic functions

- $[n] = \lambda f x . f^n x$
  - $f^n x = f(f(\cdots(f x)\cdots))$, where f is applied repeatedly n times
- Successor: $[succ] = \lambda p f x . f(p f x)$
- Addition: $[plus] = \lambda p q f x . p f(q f x)$

Encoding arithmetic functions

- $[n] = \lambda f x. f^n x$
  - $f^n x = f(f(\cdots(f x)\cdots))$, where f is applied repeatedly n times
- Successor: $[succ] = \lambda p f x. f(p f x)$
- Addition: $[plus] = \lambda p q f x. p f (q f x)$
- Multiplication: $[mult] = \lambda p q f x. p(q f)x$

Encoding arithmetic functions

- $[n] = \lambda f x . f^n x$
  - $f^n x = f(f(\cdots(f x)\cdots))$, where f is applied repeatedly n times
- Successor: $[succ] = \lambda p f x . f(p f x)$
- Addition: $[plus] = \lambda p q f x . p f(q f x)$
- Multiplication: $[mult] = \lambda p q f x . p(q f)x$
- Exponentiation: $[exp] = \lambda p q f x . q p f x$

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 $[f][n_1]\cdots[n_k] \xrightarrow{*}_{\beta} [f(n_1,\ldots,n_k)]$

- 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 [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

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

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

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

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

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

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

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

Definition

The class of primitive recursive functions is the smallest class of 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

   Zero $Z(n) = 0$

   Successor $S(n) = n + 1$

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

   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 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

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

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

Encoding recursive functions

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

Encoding recursive functions

- $[n] = \lambda f x . f^n x$
- Zero: $[Z] = \lambda x . [\mathtt{0}]$

Encoding recursive functions

- $[n] = \lambda f x. f^n x$
- Zero: $[Z] = \lambda x. [0]$
- Successor: $[succ] = \lambda p f x. f(p f x)$

Encoding recursive functions

- $[n] = \lambda f x . f^n x$
- Zero: $[Z] = \lambda x . [0]$
- Successor: $[succ] = \lambda p f x . f (p f x)$
- Projection: $\left[\Pi_i^k\right] = \lambda x_1 x_2 \cdots x_k . x_i$

Encoding recursive functions

- $[n] = \lambda f x. f^n x$
- Zero: $[Z] = \lambda x.[\mathtt{0}]$
- Successor: $[succ] = \lambda p f x. f(p f x)$
- Projection: $\left[\Pi_i^k\right] = \lambda x_1 x_2 \cdots x_k . x_i$
- Composition: If $f : \mathbb{N}^k \to \mathbb{N}$ is defined by $f = g \circ (h_1, \ldots, h_\ell)$

$$[f] = \lambda x_1 x_2 \cdots x_k . [g]\, ([h_1] x_1 x_2 \cdots x_k) \cdots ([h_\ell] x_1 x_2 \cdots x_k)$$

Encoding recursive functions

- Primitive recursion: Suppose $f(n, \vec{n})$ is defined from $g(\vec{n})$ and $h(n, m, \vec{n})$

Encoding recursive functions

- Primitive recursion: Suppose $f(n, \vec{n})$ is defined from $g(\vec{n})$ and $h(n, m, \vec{n})$
- We need to eliminate recursion

Encoding recursive functions

- Primitive recursion: Suppose $f(n, \vec{n})$ is defined from $g(\vec{n})$ and $h(n, m, \vec{n})$
- We need to eliminate recursion
  - λ-calculus functions are anonymous

Encoding recursive functions

- Primitive recursion: Suppose $f(n,\vec{n})$ is defined from $g(\vec{n})$ and $h(n,m,\vec{n})$
- We need to eliminate recursion
  - λ-calculus functions are anonymous
  - Cannot directly use name of f inside definition of f

Encoding recursive functions

- Primitive recursion: Suppose $f(n, \vec{n})$ is defined from $g(\vec{n})$ and $h(n, m, \vec{n})$
- We need to eliminate recursion
  - λ-calculus functions are anonymous
  - Cannot directly use name of f inside definition of f
- We convert recursion into iteration

Encoding recursive functions

- Primitive recursion: Suppose $f(n,\vec{n})$ is defined from $g(\vec{n})$ and $h(n,m,\vec{n})$
- We need to eliminate recursion
  - λ-calculus functions are anonymous
  - Cannot directly use name of f inside definition of f
- We convert recursion into iteration
- Given n and $\vec{n}$, generate a sequence of pairs

$$(0,a_0),(1,a_1),\ldots,(n,a_n)$$

where

Encoding recursive functions

- Primitive recursion: Suppose $f(n, \vec{n})$ is defined from $g(\vec{n})$ and $h(n, m, \vec{n})$
- We need to eliminate recursion
  - λ-calculus functions are anonymous
  - Cannot directly use name of f inside definition of f
- We convert recursion into iteration
- Given n and $\vec{n}$, generate a sequence of pairs

$$(0, a_0), (1, a_1), \ldots, (n, a_n)$$

where

- $a_0 = g(\vec{n})$

Encoding recursive functions

- Primitive recursion: Suppose $f(n, \vec{n})$ is defined from $g(\vec{n})$ and $h(n, m, \vec{n})$
- We need to eliminate recursion
  - λ-calculus functions are anonymous
  - Cannot directly use name of f inside definition of f
- We convert recursion into iteration
- Given n and $\vec{n}$, generate a sequence of pairs

$$(0, a_0), (1, a_1), \ldots, (n, a_n)$$

where

- $a_0 = g(\vec{n})$
- $a_{i+1} = h(i, a_i, \vec{n})$

Encoding recursive functions

- Primitive recursion: Suppose $f(n, \vec{n})$ is defined from $g(\vec{n})$ and $h(n, m, \vec{n})$
- We need to eliminate recursion
  - λ-calculus functions are anonymous
  - Cannot directly use name of f inside definition of f
- We convert recursion into iteration
- Given n and $\vec{n}$, generate a sequence of pairs

$$(0, a_0), (1, a_1), \ldots, (n, a_n)$$

  where
  - $a_0 = g(\vec{n})$
  - $a_{i+1} = h(i, a_i, \vec{n})$
- Finally we have $a_n = f(n, \vec{n})$

Encoding recursive functions

- Primitive recursion: Suppose $f(n, \vec{n})$ is defined from $g(\vec{n})$ and $h(n, m, \vec{n})$

Encoding recursive functions

- Primitive recursion: Suppose $f(n, \vec{n})$ is defined from $g(\vec{n})$ and $h(n, m, \vec{n})$
- Given n and $\vec{n}$, generate a sequence of pairs

$$(0, a_0), (1, a_1), \ldots, (n, a_n)$$

where $a_0 = g(\vec{n})$ and $a_{i+1} = h(i, a_i, \vec{n})$

Encoding recursive functions

- Primitive recursion: Suppose $f(n,\vec{n})$ is defined from $g(\vec{n})$ and $h(n,m,\vec{n})$
- Given n and $\vec{n}$, generate a sequence of pairs

$$(0,a_0),(1,a_1),\ldots,(n,a_n)$$

where $a_0 = g(\vec{n})$ and $a_{i+1} = h(i,a_i,\vec{n})$

- Generate the sequence by the following recursion

$$\begin{array}{lclcl}
t(0) & = & (0,a_0) & = & (0, g(\vec{n})) \\
t(i+1) & = & (i+1,a_{i+1}) & = & (i+1, h(i,a_i,\vec{n})) \\
 & & & = & (\ \mathit{succ}(\mathit{fst}(t(i))), \\
 & & & & \ \ h(\mathit{fst}(t(i)),\mathit{snd}(t(i)),\vec{n})\ )
\end{array}$$

Encoding recursive functions

- Primitive recursion: Suppose $f(n, \vec{n})$ is defined from $g(\vec{n})$ and $h(n, m, \vec{n})$
- Given n and $\vec{n}$, generate a sequence of pairs

$$(0, a_0), (1, a_1), \ldots, (n, a_n)$$

  where $a_0 = g(\vec{n})$ and $a_{i+1} = h(i, a_i, \vec{n})$
- Generate the sequence by the following recursion

$$\begin{array}{lclcl} t(0) & = & (0, a_0) & = & (0, g(\vec{n})) \\ t(i+1) & = & (i+1, a_{i+1}) & = & (i+1, h(i, a_i, \vec{n})) \\ & & & = & (\ \mathit{succ}(\mathit{fst}(t(i))), \\ & & & & \ \ h(\mathit{fst}(t(i)), \mathit{snd}(t(i)), \vec{n})\ ) \end{array}$$

  - *fst* and *snd* return the first and second components of a pair

Encoding recursive functions

- Primitive recursion: Suppose $f(n,\vec{n})$ is defined from $g(\vec{n})$ and $h(n,m,\vec{n})$
- Given n and $\vec{n}$, generate a sequence of pairs

$$(0,a_0),(1,a_1),\ldots,(n,a_n)$$

  where $a_0 = g(\vec{n})$ and $a_{i+1} = h(i,a_i,\vec{n})$
- Generate the sequence by the following recursion

$$\begin{array}{lcccl} t(0) & = & (0,a_0) & = & (0,g(\vec{n})) \\ t(i+1) & = & (i+1,a_{i+1}) & = & (i+1,h(i,a_i,\vec{n})) \\ & & & = & (\ \mathit{succ}(\mathit{fst}(t(i))), \\ & & & & \ \ h(\mathit{fst}(t(i)),\mathit{snd}(t(i)),\vec{n})\ \ ) \end{array}$$

  - *fst* and *snd* return the first and second components of a pair
  - $f(n,\vec{n})$ can be retrieved as $\mathit{snd}(t(n))$

Encoding recursive functions

- Generate the sequence by the following recursion

$$
\begin{array}{lclcl}
t(0) & = & (0, a_0) & = & (0, g(\vec{n})) \\
t(i+1) & = & (i+1, a_{i+1}) & = & (i+1, h(i, a_i, \vec{n})) \\
 & & & = & (\ \mathit{succ}(\mathit{fst}(t(i))), \\
 & & & & \ h(\mathit{fst}(t(i)), \mathit{snd}(t(i)), \vec{n})\ )
\end{array}
$$

Encoding recursive functions

- Generate the sequence by the following recursion

$$
\begin{array}{lcl}
t(0) & = & (0, a_0) & = & (0, g(\vec{n})) \\
t(i+1) & = & (i+1, a_{i+1}) & = & (i+1, h(i, a_i, \vec{n})) \\
 & & & = & (\ \mathit{succ}(\mathit{fst}(t(i))), \\
 & & & & \ \ h(\mathit{fst}(t(i)), \mathit{snd}(t(i)), \vec{n})\ )
\end{array}
$$

- We generate the $t(i)$'s by iteration

Encoding recursive functions

- Generate the sequence by the following recursion

$$
\begin{array}{lclcl}
t(0) & = & (0, a_0) & = & (0, g(\vec{n})) \\
t(i+1) & = & (i+1, a_{i+1}) & = & (i+1, h(i, a_i, \vec{n})) \\
 & & & = & (\ \mathit{succ}(\mathit{fst}(t(i))), \\
 & & & & \ \ h(\mathit{fst}(t(i)), \mathit{snd}(t(i)), \vec{n})\ )
\end{array}
$$

- We generate the $t(i)$'s by iteration
- Define $\mathit{Init} = (0, g(\vec{n}))$ and $\mathit{Step}(t(i)) = t(i+1)$

Encoding recursive functions

- Generate the sequence by the following recursion

$$
\begin{array}{lllll}
t(0) & = & (0, a_0) & = & (0, g(\vec{n})) \\
t(i+1) & = & (i+1, a_{i+1}) & = & (i+1, h(i, a_i, \vec{n})) \\
 & & & = & (\ \mathit{succ}(\mathit{fst}(t(i))), \\
 & & & & \ \ h(\mathit{fst}(t(i)), \mathit{snd}(t(i)), \vec{n})\ )
\end{array}
$$

- We generate the $t(i)$'s by iteration
- Define $\mathit{Init} = (0, g(\vec{n}))$ and $\mathit{Step}(t(i)) = t(i+1)$
- So $t(n) = \mathit{Step}^n(\mathit{Init})$ ...

Encoding recursive functions

- Generate the sequence by the following recursion

$$
\begin{array}{lclcl}
t(0) & = & (0, a_0) & = & (0, g(\vec{n})) \\
t(i+1) & = & (i+1, a_{i+1}) & = & (i+1, h(i, a_i, \vec{n})) \\
 & & & = & (\ \mathit{succ}(\mathit{fst}(t(i))), \\
 & & & & \ \ h(\mathit{fst}(t(i)), \mathit{snd}(t(i)), \vec{n})\ )
\end{array}
$$

- We generate the $t(i)$'s by iteration
- Define $\mathit{Init} = (0, g(\vec{n}))$ and $\mathit{Step}(t(i)) = t(i+1)$
- So $t(n) = \mathit{Step}^n(\mathit{Init})$ ...
- ...and $f(n, \vec{n}) = \mathit{snd}(t(n)) = \mathit{snd}(\mathit{Step}^n(\mathit{Init}))$

Encoding pairs, *fst* and *snd*

- $[pair] = \lambda xyz.zxy$

Encoding pairs, *fst* and *snd*

- $[pair] = \lambda xyz.zxy$
- $[pair]ab \xrightarrow{*}_{\beta} \lambda z.zab$

Encoding pairs, *fst* and *snd*

- $[pair] = \lambda xyz.zxy$
- $[pair]ab \xrightarrow{*}_{\beta} \lambda z.zab$
- $[fst] = \lambda p.p(\lambda xy.x)$

Encoding pairs, *fst* and *snd*

- $[pair] = \lambda xyz.zxy$
- $[pair]ab \xrightarrow{*}_{\beta} \lambda z.zab$
- $[fst] = \lambda p.p(\lambda xy.x)$
- $[fst]([pair]ab) \xrightarrow{*}_{\beta} (\lambda p.p(\lambda xy.x))(\lambda z.zab) \longrightarrow_{\beta}$
  $(\lambda z.zab)(\lambda xy.x) \longrightarrow_{\beta} (\lambda xy.x)ab \longrightarrow_{\beta} (\lambda y.a)b \longrightarrow_{\beta} a$

Encoding pairs, *fst* and *snd*

- $[pair] = \lambda xyz.zxy$
- $[pair]ab \xrightarrow{*}_\beta \lambda z.zab$
- $[fst] = \lambda p.p(\lambda xy.x)$
- $[fst]([pair]ab) \xrightarrow{*}_\beta (\lambda p.p(\lambda xy.x))(\lambda z.zab) \longrightarrow_\beta$
  $(\lambda z.zab)(\lambda xy.x) \longrightarrow_\beta (\lambda xy.x)ab \longrightarrow_\beta (\lambda y.a)b \longrightarrow_\beta a$
- $[snd] = \lambda p.(p(\lambda xy.y))$

Encoding pairs, *fst* and *snd*

- $[pair] = \lambda xyz.zxy$
- $[pair]ab \xrightarrow{*}_{\beta} \lambda z.zab$
- $[fst] = \lambda p.p(\lambda xy.x)$
- $[fst]([pair]ab) \xrightarrow{*}_{\beta} (\lambda p.p(\lambda xy.x))(\lambda z.zab) \longrightarrow_{\beta}$
  $(\lambda z.zab)(\lambda xy.x) \longrightarrow_{\beta} (\lambda xy.x)ab \longrightarrow_{\beta} (\lambda y.a)b \longrightarrow_{\beta} a$
- $[snd] = \lambda p.(p(\lambda xy.y))$
- $[snd]([pair]ab) \xrightarrow{*}_{\beta} (\lambda p.p(\lambda xy.y))(\lambda z.zab) \longrightarrow_{\beta}$
  $(\lambda z.zab)(\lambda xy.y) \longrightarrow_{\beta} (\lambda xy.y)ab \longrightarrow_{\beta} (\lambda y.y)b \longrightarrow_{\beta} b$

Encoding primitive recursion

- $Init = (0, g(\vec{n}))$

Encoding primitive recursion

- $Init = (0, g(\vec{n}))$
- $Step(t(i)) = (i+1, a_{i+1}) = (succ(fst(t(i))), h(fst(t(i)), snd(t(i)), \vec{n}))$

Encoding primitive recursion

- $\mathit{Init} = (0, g(\vec{n}))$
- $\mathit{Step}(t(i)) = (i+1, a_{i+1}) = (\mathit{succ}(\mathit{fst}(t(i))), h(\mathit{fst}(t(i)), \mathit{snd}(t(i)), \vec{n}))$
- $t(n) = \mathit{Step}^n(\mathit{Init})$ and $f(n, \vec{n}) = \mathit{snd}(t(n))$

Encoding primitive recursion

- $Init = (0, g(\vec{n}))$
- $Step(t(i)) = (i+1, a_{i+1}) = (succ(fst(t(i))), h(fst(t(i)), snd(t(i)), \vec{n}))$
- $t(n) = Step^n(Init)$ and $f(n, \vec{n}) = snd(t(n))$
- $[Init] = [pair][0]([g]\, x_1 \ldots x_k)$

Encoding primitive recursion

- $Init = (0, g(\vec{n}))$
- $Step(t(i)) = (i+1, a_{i+1}) = (succ(fst(t(i))), h(fst(t(i)), snd(t(i)), \vec{n}))$
- $t(n) = Step^n(Init)$ and $f(n, \vec{n}) = snd(t(n))$
- $[Init] = [pair][0]([g]\, x_1 \ldots x_k)$
- $[Step] = \lambda y.[pair]([succ]([fst]\, y))([h]([fst]\, y)([snd]\, y)x_1 \ldots x_k)$

Encoding primitive recursion

- $Init = (0, g(\vec{n}))$
- $Step(t(i)) = (i+1, a_{i+1}) = (succ(fst(t(i))), h(fst(t(i)), snd(t(i)), \vec{n}))$
- $t(n) = Step^n(Init)$ and $f(n, \vec{n}) = snd(t(n))$
- $[Init] = [pair][0]([g]x_1 \ldots x_k)$
- $[Step] = \lambda y.[pair]([succ]([fst]y))([h]([fst]y)([snd]y)x_1 \ldots x_k)$
  - $\vec{n}$ appears "free" in both *Init* and *Step*, so in the encodings we leave the variables $x_1, \ldots, x_k$ free

Encoding primitive recursion

- $Init = (0, g(\vec{n}))$
- $Step(t(i)) = (i+1, a_{i+1}) = (succ(fst(t(i))), h(fst(t(i)), snd(t(i)), \vec{n}))$
- $t(n) = Step^n(Init)$ and $f(n, \vec{n}) = snd(t(n))$
- $[Init] = [pair][0]([g]\, x_1 \ldots x_k)$
- $[Step] = \lambda y.[pair]([succ]([fst]\, y))([h]([fst]\, y)([snd]\, y) x_1 \ldots x_k)$
    - $\vec{n}$ appears "free" in both *Init* and *Step*, so in the encodings we leave the variables $x_1, \ldots, x_k$ free
- $[f] = \lambda x\, x_1 x_2 \cdots x_k.[snd](x\,[Step][Init])$

Encoding primitive recursion

- $[f] = \lambda x\, x_1 x_2 \cdots x_k . [snd](x\,[Step][Init])$

Encoding primitive recursion

- $[f] = \lambda x x_1 x_2 \cdots x_k . [snd](x\,[Step][Init])$
- $[f][n][n_1]\cdots[n_k] \xrightarrow{*}_\beta [snd]([n]\big[Step'\big]\big[Init'\big]) \xrightarrow{*}_\beta$ $[snd](\big[Step'\big]^n \big[Init'\big])$

Encoding primitive recursion

- $[f] = \lambda x x_1 x_2 \cdots x_k . [snd](x\,[Step]\,[Init])$
- $[f][n][n_1]\cdots[n_k] \xrightarrow{*}_\beta [snd]([n]\left[Step'\right]\left[Init'\right]) \xrightarrow{*}_\beta$ $[snd](\left[Step'\right]^n\left[Init'\right])$
    - $[Init'] = [Init][x_1 := [n_1], \ldots, x_k := [n_k]]$

Encoding primitive recursion

- $[f] = \lambda x x_1 x_2 \cdots x_k . [snd](x\,[Step]\,[Init])$
- $[f][n][n_1]\cdots[n_k] \xrightarrow{*}_{\beta} [snd]([n]\left[Step'\right]\left[Init'\right]) \xrightarrow{*}_{\beta}$ $[snd](\left[Step'\right]^n \left[Init'\right])$
    - $[Init'] = [Init][x_1 := [n_1], \ldots, x_k := [n_k]]$
    - $[Step'] = [Step][x_1 := [n_1], \ldots, x_k := [n_k]]$

- $[f] = \lambda x x_1 x_2 \cdots x_k . [snd](x\,[Step][Init])$
- $[f][n][n_1]\cdots[n_k] \xrightarrow{*}_{\beta} [snd]([n]\left[Step'\right]\left[Init'\right]) \xrightarrow{*}_{\beta}$ $[snd](\left[Step'\right]^n \left[Init'\right])$
  - $[Init'] = [Init][x_1 := [n_1], \ldots, x_k := [n_k]]$
  - $[Step'] = [Step][x_1 := [n_1], \ldots, x_k := [n_k]]$
- Check that $\left[Step'\right]([pair][i][f(i,\vec{n})]) \xrightarrow{*}_{\beta} [pair][i+1][f(i+1,\vec{n})]$

Encoding primitive recursion

- Check that $\left[Step'\right]([pair][i][f(i,\vec{n})]) \xrightarrow{*}_{\beta} [pair][i+1][f(i+1,\vec{n})]$

Encoding primitive recursion

- Check that $\left[Step'\right]([pair][i][f(i,\vec{n})]) \xrightarrow{*}_{\beta} [pair][i+1][f(i+1,\vec{n})]$
- $\left[Step'\right]([pair][i][f(i,\vec{n})])$

$$
\begin{array}{ll}
\xrightarrow{*}_{\beta} & [pair]\ ([succ]([fst]))([pair][i][f(i,\vec{n})]) \\
& \quad ([h]\ ([fst]([pair][i][f(i,\vec{n})])) \\
& \qquad ([snd]([pair][i][f(i,\vec{n})])) \\
& \qquad [n_1]\cdots[n_k]) \\
\xrightarrow{*}_{\beta} & [pair]\ ([succ][i])\ ([h][i][f(i,\vec{n})][n_1]\cdots[n_k]) \\
\xrightarrow{*}_{\beta} & [pair]\ [i+1]\ [h(i,f(i,\vec{n}),\vec{n})] \\
\xrightarrow{*}_{\beta} & [pair]\ [i+1]\ [f(i+1,\vec{n})])
\end{array}
$$

Encoding primitive recursion

- Check that $\left[Step'\right]^i \left[Init'\right] \xrightarrow{*}_\beta [pair]\ [i]\ [f(i,\vec{n})]$

Encoding primitive recursion

- Check that $[Step']^i\,[Init'] \xrightarrow{*}_\beta [pair]\ [i]\ [f(i,\vec{n})]$
- $[Step']^0\,[Init'] \xrightarrow{*}_\beta [Init'] = [pair]\ [0]\ [g(\vec{n})] = [pair]\ [0]\ [f(0,\vec{n})]$

Encoding primitive recursion

- Check that $[Step']^i\,[Init'] \xrightarrow{*}_\beta [pair]\ [i]\ [f(i,\vec{n})]$
- $[Step']^0\,[Init'] \xrightarrow{*}_\beta [Init'] = [pair]\ [0]\ [g(\vec{n})] = [pair]\ [0]\ [f(0,\vec{n})]$
- $$\begin{aligned} [Step']^{i+1}\,[Init'] &= [Step']([Step']^i\,[Init']) \\ &\xrightarrow{*}_\beta [Step']([pair]\ [i]\ [f(i,\vec{n})]) \quad \text{(ind. hyp.)} \\ &\xrightarrow{*}_\beta [pair]\ [i+1]\ [f(i+1,\vec{n})] \end{aligned}$$

Encoding primitive recursion

- $[f] = \lambda x\, x_1 x_2 \cdots x_k . [snd](x\,[Step][Init])$

Encoding primitive recursion

- $[f] = \lambda x x_1 x_2 \cdots x_k . [snd](x\,[Step]\,[Init])$
- $[f][n][n_1]\cdots[n_k] \xrightarrow{*}_{\beta} [snd]([n]\left[Step'\right]\left[Init'\right]) \xrightarrow{*}_{\beta}$ $[snd](\left[Step'\right]^n \left[Init'\right])$

Encoding primitive recursion

- $[f] = \lambda x x_1 x_2 \cdots x_k . [snd](x\,[Step][Init])$
- $[f][n][n_1]\cdots[n_k] \xrightarrow{*}_\beta [snd]([n]\left[Step'\right]\left[Init'\right]) \xrightarrow{*}_\beta [snd](\left[Step'\right]^n\left[Init'\right])$
- $\left[Step'\right]^n\left[Init'\right] \xrightarrow{*}_\beta [pair]\ [n]\ [f(n,\vec{n})]$

Encoding primitive recursion

- $[f] = \lambda x x_1 x_2 \cdots x_k.[snd](x\,[Step][Init])$
- $[f][n][n_1]\cdots[n_k] \xrightarrow{*}_\beta [snd]([n]\left[Step'\right]\left[Init'\right]) \xrightarrow{*}_\beta$ $[snd](\left[Step'\right]^n \left[Init'\right])$
- $\left[Step'\right]^n \left[Init'\right] \xrightarrow{*}_\beta [pair]\ [n]\ [f(n,\vec{n})]$
- So
  $[f][n][n_1]\cdots[n_k] \xrightarrow{*}_\beta [snd]([pair]\ [n]\ [f(n,\vec{n})]) \xrightarrow{*}_\beta [f(n,\vec{n})]$

Encoding primitive recursion

- $[f] = \lambda x x_1 x_2 \cdots x_k . [snd](x\,[Step][Init])$
- $[f][n][n_1] \cdots [n_k] \xrightarrow{*}_{\beta} [snd]([n][Step'][Init']) \xrightarrow{*}_{\beta}$ $[snd]([Step']^n [Init'])$
- $[Step']^n [Init'] \xrightarrow{*}_{\beta} [pair]\ [n]\ [f(n, \vec{n})]$
- So $[f][n][n_1] \cdots [n_k] \xrightarrow{*}_{\beta} [snd]([pair]\ [n]\ [f(n, \vec{n})]) \xrightarrow{*}_{\beta} [f(n, \vec{n})]$
- The expression $[PR]$ encodes the schema of primitive recursion

$$\begin{aligned}[PR] = \lambda h\, g\, x\, x_1 \cdots x_k . [snd](x(\lambda y . [pair]\ & ([succ]([fst]\, y)) \\ & (h([fst]\, y)([snd]\, y) x_1 \ldots x_k))) \\ & ([pair][0](g\, x_1 \ldots x_k))\end{aligned}$$