Programming Language Concepts: Lecture 17

S P Suresh

March 15, 2021

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**: $[mult] = \lambda p q f . p(q f)$

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**: $[mult] = \lambda p q f. p(q f)$
- **Exponentiation**: $[exp] = \lambda p q. p q$

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

Definition

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

Definition

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

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

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

Recursive functions

Definition

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

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

Definition

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

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

Recursive functions

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**

Recursive functions

Definition

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

1. containing the **initial functions**

   Zero $Z(n) = 0$

Recursive functions

Definition

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

1. containing the **initial functions**

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

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

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

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

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

Recursive functions

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 **(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**: $[Z] = \lambda x. [0]$

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**: $[Z] = \lambda x. [0]$
- **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 $f = g \circ (h_1, \ldots, h_l)$

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

Encoding recursive functions

- **Primitive recursion**: Suppose f is defined via primitive recursion from g and h

Encoding recursive functions

- **Primitive recursion**: Suppose f is defined via primitive recursion from g and h
- We need to eliminate recursion

Encoding recursive functions

- **Primitive recursion**: Suppose f is defined via primitive recursion from g and h
- We need to eliminate recursion
  - λ-calculus functions are anonymous

Encoding recursive functions

- **Primitive recursion**: Suppose f is defined via primitive recursion from g and h
- 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 is defined via primitive recursion from g and h
- 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 is defined via primitive recursion from g and h
- 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 l and $\vec{n}$, generate a sequence of pairs

$$(0, a_0), (1, a_1), \ldots, (l, a_l)$$

where

Encoding recursive functions

- **Primitive recursion**: Suppose f is defined via primitive recursion from g and h
- 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 l and $\vec{n}$, generate a sequence of pairs

$$(0, a_0), (1, a_1), \ldots, (l, a_l)$$

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

Encoding recursive functions

- **Primitive recursion**: Suppose f is defined via primitive recursion from g and h
- 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 l and $\vec{n}$, generate a sequence of pairs

$$(0, a_0), (1, a_1), \ldots, (l, a_l)$$

where

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

Encoding recursive functions

- **Primitive recursion**: Suppose f is defined via primitive recursion from g and h
- 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 l and $\vec{n}$, generate a sequence of pairs

$$(0, a_0), (1, a_1), \ldots, (l, a_l)$$

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

Encoding recursive functions

- **Primitive recursion**: Suppose f is defined via primitive recursion from g and h

Encoding recursive functions

- **Primitive recursion**: Suppose f is defined via primitive recursion from g and h
- Given l and $\vec{n}$, generate a sequence of pairs

$$(0, a_0), (1, a_1), \ldots, (l, a_l)$$

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

Encoding recursive functions

- **Primitive recursion**: Suppose f is defined via primitive recursion from g and h
- Given l and $\vec{n}$, generate a sequence of pairs

$$(0, a_0), (1, a_1), \ldots, (l, a_l)$$

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

- Generate the sequence by the following recursion

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

Encoding recursive functions

- **Primitive recursion**: Suppose f is defined via primitive recursion from g and h
- Given l and $\vec{n}$, generate a sequence of pairs

$$(0, a_0), (1, a_1), \ldots, (l, a_l)$$

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

- Generate the sequence by the following recursion

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

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

Encoding recursive functions

- **Primitive recursion**: Suppose f is defined via primitive recursion from g and h
- Given l and $\vec{n}$, generate a sequence of pairs

$$(0, a_0), (1, a_1), \ldots, (l, a_l)$$

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

- Generate the sequence by the following recursion

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

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

Encoding recursive functions

- Generate the sequence by the following recursion

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

Encoding recursive functions

- Generate the sequence by the following recursion

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

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

Encoding recursive functions

- Generate the sequence by the following recursion

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

- 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

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

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

Encoding recursive functions

- Generate the sequence by the following recursion

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

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

Encoding pairs, *fst* and *snd*

- $[pair] = \lambda x\, y\, z.\, z\, x\, y$

Encoding pairs, *fst* and *snd*

- $[pair] = \lambda x\, y\, z. z\, x\, y$
- $[pair]\, a\, b \xrightarrow{*}_{\beta} \lambda z. z\, a\, b$

Encoding pairs, *fst* and *snd*

- $[pair] = \lambda x\, y\, z.\, z\, x\, y$
- $[pair]\, a\, b \xrightarrow{*}_{\beta} \lambda z.\, z\, a\, b$
- $[fst] = \lambda p.\, p(\lambda x\, y.x)$

Encoding pairs, *fst* and *snd*

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

Encoding pairs, *fst* and *snd*

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

Encoding pairs, *fst* and *snd*

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

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

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

Encoding primitive recursion

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

Encoding primitive recursion

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

Encoding primitive recursion

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

Encoding primitive recursion

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

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

Encoding primitive recursion

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

Encoding primitive recursion

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

Encoding primitive recursion

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

Encoding primitive recursion

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

Encoding primitive recursion

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

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

Encoding primitive recursion

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

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

Encoding primitive recursion

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

Encoding primitive recursion

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

Encoding primitive recursion

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

Encoding primitive recursion

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

Encoding primitive recursion

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

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