Programming Language Concepts: Lecture 18

S P Suresh

March 17, 2021

Encoding primitive recursion

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

Encoding primitive recursion

- $[f] = \lambda x\, x_1\, x_2 \cdots x_k . [snd](x\, [Step]\, [Init])$
  - $[Init] = [pair]\, [0]\, ([g]\, x_1 \ldots x_k)$

Encoding primitive recursion

- $[f] = \lambda x\, x_1 x_2 \cdots x_k . [snd](x\,[Step][Init])$
  - $[Init] = [pair][0]([g]\, x_1 \ldots x_k)$
  - $[Step] = \lambda y.[pair]([succ]([fst]\, y))([h]([fst]\, y)([snd]\, y) x_1 \ldots x_k)$

Encoding primitive recursion

- $[f] = \lambda x\, x_1 x_2 \cdots x_k . [snd](x\, [Step][Init])$
  - $[Init] = [pair][0]([g]\, x_1 \ldots x_k)$
  - $[Step] = \lambda y . [pair]([succ]([fst]\, y))([h]([fst]\, y)([snd]\, y) x_1 \ldots x_k)$
- $[f][l][n_1] \cdots [n_k] \xrightarrow{*}_\beta [f(l, \vec{n})]$

Encoding primitive recursion

- $[f] = \lambda x\, x_1 x_2 \cdots x_k. [snd](x\,[Step][Init])$
  - $[Init] = [pair][0]([g]\, x_1 \ldots x_k)$
  - $[Step] = \lambda y. [pair]([succ]([fst]\, y))([h]([fst]\, y)([snd]\, y) x_1 \ldots x_k)$
- $[f][l][n_1] \cdots [n_k] \xrightarrow{*}_{\beta} [f(l, \vec{n})]$
- The expression $[PR]$ encodes the schema of primitive recursion

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

Encoding μ-recursion

- $f(\vec{n}) = \mu i.(g(i, \vec{n}) = 0)$ can be expressed as the following (potentially unbounded) `while` loop:

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

Encoding μ-recursion

- $f(\vec{n}) = \mu i.(g(i, \vec{n}) = 0)$ can be expressed as the following (potentially unbounded) **while** loop:

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

- Implement the **while** loop using recursion:

```
int searchFrom(i, n1, n2, ..., nk) {
    if (iszero(g(i, n1, n2, ..., nk))) return n;
    else return searchFrom(i+1, n1, n2, ..., nk);
}
f(n1, n2, ..., nk) = searchFrom(0, n1, n2, ..., nk);
```

Encoding booleans

- Need ways to encode booleans, if-then-else, test for zero and recursive definitions in λ-calculus
- $[true] = \lambda x\, y. x$

Encoding booleans

- Need ways to encode booleans, if-then-else, test for zero and recursive definitions in λ-calculus
- $[\mathit{true}] = \lambda x\, y. x$
- $[\mathit{false}] = \lambda x\, y. y$

Encoding booleans

- Need ways to encode booleans, if-then-else, test for zero and recursive definitions in λ-calculus
- $[\mathit{true}] = \lambda x\, y.x$
- $[\mathit{false}] = \lambda x\, y.y$
- $[\mathit{if\text{-}then\text{-}else}] = \lambda b\, x\, y.b\, x\, y$

Encoding booleans

- Need ways to encode booleans, if-then-else, test for zero and recursive definitions in λ-calculus
- $[\mathit{true}] = \lambda x\, y. x$
- $[\mathit{false}] = \lambda x\, y. y$
- $[\mathit{if\text{-}then\text{-}else}] = \lambda b\, x\, y. b\, x\, y$
- **Syntactic sugar**: $[\mathit{if\text{-}then\text{-}else}]\ b\ f\ g$ is written as $\mathit{if}\ b\ \mathit{then}\ f\ \mathit{else}\ g$

Encoding booleans

- Need ways to encode booleans, if-then-else, test for zero and recursive definitions in λ-calculus
- $[\mathit{true}] = \lambda x\, y.x$
- $[\mathit{false}] = \lambda x\, y.y$
- $[\mathit{if\text{-}then\text{-}else}] = \lambda b\, x\, y.\, b\, x\, y$
- **Syntactic sugar**: $[\mathit{if\text{-}then\text{-}else}]\; b\; f\; g$ is written as $\mathit{if}\; b\; \mathit{then}\; f\; \mathit{else}\; g$

$$\begin{array}{rcl}
\mathit{if}\,[\mathit{true}]\;\mathit{then}\; f\; \mathit{else}\; g & = & \\
(\lambda b\, x\, y.b\, x\, y)(\lambda x\, y.x) f\, g & \longrightarrow_\beta & (\lambda x\, y.(\lambda x\, y.x) x\, y) f\, g \\
 & \longrightarrow_\beta & (\lambda y.(\lambda x\, y.x) f\, y) g \\
 & \longrightarrow_\beta & (\lambda x\, y.x) f\, g \longrightarrow_\beta (\lambda y.f) g \longrightarrow_\beta f
\end{array}$$

Encoding booleans

- Need ways to encode booleans, if-then-else, test for zero and recursive definitions in λ-calculus
- $[\mathit{true}] = \lambda x\,y.x$
- $[\mathit{false}] = \lambda x\,y.y$
- $[\mathit{if\text{-}then\text{-}else}] = \lambda b\,x\,y.b\,x\,y$
- **Syntactic sugar**: $[\mathit{if\text{-}then\text{-}else}]\ b\ f\ g$ is written as $\mathit{if}\ b\ \mathit{then}\ f\ \mathit{else}\ g$

$$
\begin{array}{rcl}
\mathit{if}\,[\mathit{true}]\ \mathit{then}\ f\ \mathit{else}\ g & = & \\
(\lambda b\,x\,y.b\,x\,y)(\lambda x\,y.x)f\,g & \longrightarrow_\beta & (\lambda x\,y.(\lambda x\,y.x)x\,y)f\,g \\
 & \longrightarrow_\beta & (\lambda y.(\lambda x\,y.x)f\,y)g \\
 & \longrightarrow_\beta & (\lambda x\,y.x)f\,g \longrightarrow_\beta (\lambda y.f)g \longrightarrow_\beta f \\
\mathit{if}\,[\mathit{false}]\ \mathit{then}\ f\ \mathit{else}\ g & = & \\
(\lambda b\,x\,y.b\,x\,y)(\lambda x\,y.y)f\,g & \overset{*}{\longrightarrow}_\beta & (\lambda x\,y.y)f\,g \longrightarrow_\beta (\lambda y.y)g \longrightarrow_\beta g
\end{array}
$$

Encoding test for zero

- $[iszero] = \lambda x.x(\lambda z.[false])[true]$

Encoding test for zero

- $[iszero] = \lambda x.x(\lambda z.[false])[true]$
- $[iszero][n] \longrightarrow_\beta [n](\lambda z.[false])[true] \xrightarrow{*}_\beta (\lambda z.[false])^n[true]$

Encoding test for zero

- $[iszero] = \lambda x.x(\lambda z.[false])[true]$
- $[iszero][n] \longrightarrow_\beta [n](\lambda z.[false])[true] \xrightarrow{*}_\beta (\lambda z.[false])^n[true]$
- $(\lambda z.[false])^0[true] = [true]$

Encoding test for zero

- $[iszero] = \lambda x.x(\lambda z.[false])[true]$
- $[iszero][n] \longrightarrow_\beta [n](\lambda z.[false])[true] \xrightarrow{*}_\beta (\lambda z.[false])^n [true]$
- $(\lambda z.[false])^0 [true] = [true]$
- For $n > 0$, $(\lambda z.[false])^n [true] = (\lambda z.[false])((\lambda z.[false])^{n-1} [true]) \longrightarrow_\beta [false]$

Encoding test for zero

- $[iszero] = \lambda x.x(\lambda z.[false])[true]$
- $[iszero][n] \longrightarrow_\beta [n](\lambda z.[false])[true] \xrightarrow{*}_\beta (\lambda z.[false])^n [true]$
- $(\lambda z.[false])^0 [true] = [true]$
- For $n > 0$, $(\lambda z.[false])^n [true] = (\lambda z.[false])((\lambda z.[false])^{n-1} [true]) \longrightarrow_\beta [false]$
- Thus

Encoding test for zero

- $[iszero] = \lambda x.x(\lambda z.[false])[true]$
- $[iszero][n] \longrightarrow_\beta [n](\lambda z.[false])[true] \stackrel{*}{\longrightarrow}_\beta (\lambda z.[false])^n [true]$
- $(\lambda z.[false])^0 [true] = [true]$
- For $n > 0$, $(\lambda z.[false])^n [true] = (\lambda z.[false])((\lambda z.[false])^{n-1} [true]) \longrightarrow_\beta [false]$
- Thus
    - $[iszero][0] \stackrel{*}{\longrightarrow}_\beta [true]$

Encoding test for zero

- $[iszero] = \lambda x.x(\lambda z.[false])[true]$
- $[iszero][n] \longrightarrow_\beta [n](\lambda z.[false])[true] \xrightarrow{*}_\beta (\lambda z.[false])^n [true]$
- $(\lambda z.[false])^0 [true] = [true]$
- For $n > 0$, $(\lambda z.[false])^n [true] = (\lambda z.[false])((\lambda z.[false])^{n-1} [true]) \longrightarrow_\beta [false]$
- Thus
    - $[iszero][0] \xrightarrow{*}_\beta [true]$
    - $[iszero][n] \xrightarrow{*}_\beta [false]$ for $n > 0$

Encoding μ-recursion

- $f(\vec{n}) = \mu n.(g(n, \vec{n}) = 0)$ is expressed as follows:

```
int searchFrom(i, n1, n2, ..., nk) {
    if (iszero(g(i, n1, n2, ..., nk))) return n;
    else return searchFrom(i+1, n1, n2, ..., nk);
}
f(n1, n2, ..., nk) = searchFrom(0, n1, n2, ..., nk);
```

Encoding μ-recursion

- $f(\vec{n}) = \mu n.(g(n,\vec{n}) = 0)$ is expressed as follows:

```
int searchFrom(i, n1, n2, ..., nk) {
    if (iszero(g(i, n1, n2, ..., nk))) return n;
    else return searchFrom(i+1, n1, n2, ..., nk);
}
f(n1, n2, ..., nk) = searchFrom(0, n1, n2, ..., nk);
```

- Define $W = \lambda y.if([iszero]([g]\, y\, x_1 \cdots x_k))\, then\, (\lambda w.y)\, else\, (\lambda w.w([succ]\, y)w)$

Encoding μ-recursion

- $f(\vec{n}) = \mu n.(g(n, \vec{n}) = 0)$ is expressed as follows:

```
int searchFrom(i, n1, n2, ..., nk) {
    if (iszero(g(i, n1, n2, ..., nk))) return n;
    else return searchFrom(i+1, n1, n2, ..., nk);
}
f(n1, n2, ..., nk) = searchFrom(0, n1, n2, ..., nk);
```

- Define $W = \lambda y.if([iszero]([g]\, y x_1 \cdots x_k))\, then\, (\lambda w.y)\, else\, (\lambda w.w([succ]\, y)w)$
    - $x_1, \ldots, x_n$ occur free in W

Encoding μ-recursion

- $f(\vec{n}) = \mu n.(g(n, \vec{n}) = 0)$ is expressed as follows:

```
int searchFrom(i, n1, n2, ..., nk) {
    if (iszero(g(i, n1, n2, ..., nk))) return n;
    else return searchFrom(i+1, n1, n2, ..., nk);
}
f(n1, n2, ..., nk) = searchFrom(0, n1, n2, ..., nk);
```

- Define $W = \lambda y.\mathit{if}([\mathit{iszero}]([g]\, y\, x_1 \cdots x_k))\ \mathit{then}\ (\lambda w.y)\ \mathit{else}\ (\lambda w.w([\mathit{succ}]\, y)w)$
  - $x_1, \ldots, x_n$ occur free in W
- $[f] = \lambda x_1 \cdots x_k.W\,[0]\,W$

Encoding μ-recursion

- $f(\vec{n}) = \mu n.(g(n, \vec{n}) = 0)$ is expressed as follows:

```
int searchFrom(i, n1, n2, ..., nk) {
    if (iszero(g(i, n1, n2, ..., nk))) return n;
    else return searchFrom(i+1, n1, n2, ..., nk);
}
f(n1, n2, ..., nk) = searchFrom(0, n1, n2, ..., nk);
```

- Define $W = \lambda y.if([iszero]([g]\, y x_1 \cdots x_k))\, then\, (\lambda w.y)\, else\, (\lambda w.w([succ]\, y)w)$
  - $x_1, \ldots, x_n$ occur free in W
- $[f] = \lambda x_1 \cdots x_k.W\,[0]\,W$
- $[f][n_1]\cdots[n_k] \xrightarrow{*}_{\beta} W'\,[0]\,W'$

Encoding μ-recursion

- $f(\vec{n}) = \mu n.(g(n, \vec{n}) = 0)$ is expressed as follows:

```
int searchFrom(i, n1, n2, ..., nk) {
    if (iszero(g(i, n1, n2, ..., nk))) return n;
    else return searchFrom(i+1, n1, n2, ..., nk);
}
f(n1, n2, ..., nk) = searchFrom(0, n1, n2, ..., nk);
```

- Define $W = \lambda y.\mathit{if}([\mathit{iszero}]([g]\, y\, x_1 \cdots x_k))\ \mathit{then}\ (\lambda w.y)\ \mathit{else}\ (\lambda w.w([\mathit{succ}]\, y)w)$
  - $x_1, \ldots, x_n$ occur free in W
- $[f] = \lambda x_1 \cdots x_k.W\, [0]\, W$
- $[f][n_1] \cdots [n_k] \xrightarrow{*}_{\beta} W'\, [0]\, W'$
  - $W' = W[x_1 := [n_1], \ldots, x_k := [n_k]]$

Encoding μ-recursion

- Define $W = \lambda y.if([iszero]([g]\, y\, x_1 \cdots x_k))\; then\; (\lambda w.y)\; else\; (\lambda w.w([succ]\, y)w)$

Encoding μ-recursion

- Define $W = \lambda y.\mathit{if}([\mathit{iszero}]([g]\, y\, x_1 \cdots x_k))\ \mathit{then}\ (\lambda w.y)\ \mathit{else}\ (\lambda w.w([\mathit{succ}]\, y)w)$
- $[f] = \lambda x_1 \cdots x_k.W\,[0]\,W$

Encoding μ-recursion

- Define $W = \lambda y.if([iszero]([g]\, y\, x_1 \cdots x_k))\ then\ (\lambda w.y)\ else\ (\lambda w.w([succ]\, y)w)$
- $[f] = \lambda x_1 \cdots x_k . W\, [0]\, W$
- $[f][n_1]\cdots[n_k] \xrightarrow{*}_{\beta} W'\,[0]\,W'$

Encoding μ-recursion

- Define $W = \lambda y.if([iszero]([g]\,y\,x_1 \cdots x_k))\ then\ (\lambda w.y)\ else\ (\lambda w.w([succ]\,y)w)$
- $[f] = \lambda x_1 \cdots x_k.W\,[0]\,W$
- $[f][n_1]\cdots[n_k] \xrightarrow{*}_{\beta} W'\,[0]\,W'$
- Suppose $g(i, \vec{n}) = 0$

Encoding μ-recursion

- Define $W = \lambda y.if([iszero]([g]\, y\, x_1 \cdots x_k))\; then\; (\lambda w.y)\; else\; (\lambda w.w([succ]\, y)w)$
- $[f] = \lambda x_1 \cdots x_k . W\,[0]\,W$
- $[f][n_1]\cdots[n_k] \xrightarrow{*}_{\beta} W'\,[0]\,W'$
- Suppose $g(i, \vec{n}) = 0$
    - Then $[g][i][n_1]\cdots[n_k] \xrightarrow{*}_{\beta} [0]$

Encoding μ-recursion

- Define $W = \lambda y.if([iszero]([g]\,y\,x_1 \cdots x_k))\ then\ (\lambda w.y)\ else\ (\lambda w.w([succ]\,y)w)$
- $[f] = \lambda x_1 \cdots x_k.W\,[0]\,W$
- $[f][n_1]\cdots[n_k] \xrightarrow{*}_{\beta} W'[0]\,W'$
- Suppose $g(i,\vec{n}) = 0$
    - Then $[g][i][n_1]\cdots[n_k] \xrightarrow{*}_{\beta} [0]$
    - So $[iszero]([g][i][n_1]\cdots[n_k]) \xrightarrow{*}_{\beta} [iszero][0] \xrightarrow{*}_{\beta} [true]$

Encoding μ-recursion

- Define $W = \lambda y.\mathit{if}\,([\mathit{iszero}]([g]\,y\,x_1 \cdots x_k))\ \mathit{then}\ (\lambda w.y)\ \mathit{else}\ (\lambda w.w([\mathit{succ}]\,y)w)$
- $[f] = \lambda x_1 \cdots x_k.W\,[0]\,W$
- $[f][n_1]\cdots[n_k] \xrightarrow{*}_{\beta} W'\,[0]\,W'$
- Suppose $g(i,\vec{n}) = 0$
  - Then $[g][i][n_1]\cdots[n_k] \xrightarrow{*}_{\beta} [0]$
  - So $[\mathit{iszero}]([g][i][n_1]\cdots[n_k]) \xrightarrow{*}_{\beta} [\mathit{iszero}][0] \xrightarrow{*}_{\beta} [\mathit{true}]$
  - So

$$
\begin{aligned}
W'[i]\,W' &\xrightarrow{*}_{\beta} (\mathit{if}\,([\mathit{iszero}]([g][i][n_1]\cdots[n_k])) \\
&\qquad \mathit{then}\ (\lambda w.[i]) \\
&\qquad \mathit{else}\ (\lambda w.w([\mathit{succ}][i])w))\quad W' \\
&\xrightarrow{*}_{\beta} (\mathit{if}\,[\mathit{true}]\ \mathit{then}\ (\lambda w.[i])\ \mathit{else}\ (\lambda w.w([\mathit{succ}][i])w))\ W' \\
&\xrightarrow{*}_{\beta} (\lambda w.[i])W' \\
&\xrightarrow{*}_{\beta} [i]
\end{aligned}
$$

Encoding μ-recursion

- Suppose $g(i, \vec{n}) = m > 0$

Encoding μ-recursion

- Suppose $g(i, \vec{n}) = m > 0$
  - Then $[g][i][n_1]\cdots[n_k] \xrightarrow{*}_{\beta} [m]$

Encoding μ-recursion

- Suppose $g(i, \vec{n}) = m > 0$
  - Then $[g][i][n_1] \cdots [n_k] \xrightarrow{*}_\beta [m]$
  - So $[\mathit{iszero}]([g][i][n_1] \cdots [n_k]) \xrightarrow{*}_\beta [\mathit{iszero}][0] \xrightarrow{*}_\beta [\mathit{false}]$

Encoding μ-recursion

- Suppose $g(i, \vec{n}) = m > 0$
  - Then $[g][i][n_1]\cdots[n_k] \xrightarrow{*}_\beta [m]$
  - So $[\mathit{iszero}]([g][i][n_1]\cdots[n_k]) \xrightarrow{*}_\beta [\mathit{iszero}][0] \xrightarrow{*}_\beta [\mathit{false}]$
  - So

$$
\begin{aligned}
W'[i]W' \quad & \xrightarrow{*}_\beta \quad (\mathit{if}([\mathit{iszero}]([g][i][n_1]\cdots[n_k])) \\
& \qquad \mathit{then}\ (\lambda w.[i]) \\
& \qquad \mathit{else}\ (\lambda w.w([\mathit{succ}][i])w)) \\
& \qquad W' \\
& \xrightarrow{*}_\beta \quad (\mathit{if}[\mathit{false}]\ \mathit{then}\ (\lambda w.[i])\ \mathit{else}\ (\lambda w.w([\mathit{succ}][i])w))\ W' \\
& \xrightarrow{*}_\beta \quad (\lambda w.w([\mathit{succ}][i])w))W' \\
& \xrightarrow{*}_\beta \quad W'([\mathit{succ}][i])W' \\
& \xrightarrow{*}_\beta \quad W'[i+1]W'
\end{aligned}
$$

Encoding μ-recursion

- If $g(i, \vec{n}) = 0$ then $W' \, [i] \, W' \xrightarrow{*}_{\beta} [i]$

Encoding μ-recursion

- If $g(i, \vec{n}) = 0$ then $W' \, [i] \, W' \xrightarrow{*}_{\beta} [i]$
- If $g(i, \vec{n}) > 0$ then $W' \, [i] \, W' \xrightarrow{*}_{\beta} W' \, [i+1] \, W'$

Encoding μ-recursion

- If $g(i, \vec{n}) = 0$ then $W'\,[i]\,W' \xrightarrow{*}_{\beta} [i]$
- If $g(i, \vec{n}) > 0$ then $W'\,[i]\,W' \xrightarrow{*}_{\beta} W'\,[i+1]\,W'$
- Suppose now that $g(b, \vec{n}) = 0$ and $g(a, \vec{n}) > 0$ for all $a < b$

Encoding μ-recursion

- If $g(i, \vec{n}) = 0$ then $W' \, [i] \, W' \xrightarrow{*}_\beta [i]$
- If $g(i, \vec{n}) > 0$ then $W' \, [i] \, W' \xrightarrow{*}_\beta W' \, [i+1] \, W'$
- Suppose now that $g(b, \vec{n}) = 0$ and $g(a, \vec{n}) > 0$ for all $a < b$
- $W' \, [0] \, W' \xrightarrow{*}_\beta W' \, [1] \, W' \xrightarrow{*}_\beta W' \, [2] \, W' \xrightarrow{*}_\beta \ldots \xrightarrow{*}_\beta W' \, [b] \, W' \xrightarrow{*}_\beta [b]$

Encoding μ-recursion

- If $g(i, \vec{n}) = 0$ then $W' [i] W' \xrightarrow{*}_\beta [i]$
- If $g(i, \vec{n}) > 0$ then $W' [i] W' \xrightarrow{*}_\beta W' [i+1] W'$
- Suppose now that $g(b, \vec{n}) = 0$ and $g(a, \vec{n}) > 0$ for all $a < b$
- $W' [0] W' \xrightarrow{*}_\beta W' [1] W' \xrightarrow{*}_\beta W' [2] W' \xrightarrow{*}_\beta \ldots \xrightarrow{*}_\beta W' [b] W' \xrightarrow{*}_\beta [b]$
- Thus $[f][n_1] \cdots [n_k] \xrightarrow{*}_\beta W' [b] W' \xrightarrow{*}_\beta [b] = [\mu i . g(i, \vec{n}) = 0]$

Encoding μ-recursion

- If $g(i,\vec{n}) = 0$ then $W' \, [i] \, W' \xrightarrow{*}_\beta [i]$
- If $g(i,\vec{n}) > 0$ then $W' \, [i] \, W' \xrightarrow{*}_\beta W' \, [i+1] \, W'$
- Suppose now that $g(b,\vec{n}) = 0$ and $g(a,\vec{n}) > 0$ for all $a < b$
- $W' \, [0] \, W' \xrightarrow{*}_\beta W' \, [1] \, W' \xrightarrow{*}_\beta W' \, [2] \, W' \xrightarrow{*}_\beta \ldots \xrightarrow{*}_\beta W' \, [b] \, W' \xrightarrow{*}_\beta [b]$
- Thus $[f][n_1] \cdots [n_k] \xrightarrow{*}_\beta W' \, [b] \, W' \xrightarrow{*}_\beta [b] = [\mu i . g(i,\vec{n}) = 0]$
- The expression $[Mu] = \lambda g \, x_1 \cdots x_k . U \, [0] \, U$ encodes the schema of μ-recursion where

$$U = \lambda y.\mathit{if}([\mathit{iszero}](g \, y \, x_1 \cdots x_k)) \; \mathit{then} \; (\lambda w.y) \; \mathit{else} \; (\lambda w.w([\mathit{succ}] \, y) w)$$

Recursive definitions

- The function $f(\vec{n}) = \mu n.(g(n, \vec{n}) = 0)$ is expressed as follows:

```
int searchFrom(i, n1, n2, ..., nk) {
    if (iszero(g(i, n1, n2, ..., nk))) return n;
    else return searchFrom(i+1, n1, n2, ..., nk);
}
f(n1, n2, ..., nk) = searchFrom(0, n1, n2, ..., nk);
```

Recursive definitions

- The function $f(\vec{n}) = \mu n.(g(n,\vec{n}) = 0)$ is expressed as follows:

```
int searchFrom(i, n1, n2, ..., nk) {
    if (iszero(g(i, n1, n2, ..., nk))) return n;
    else return searchFrom(i+1, n1, n2, ..., nk);
}
f(n1, n2, ..., nk) = searchFrom(0, n1, n2, ..., nk);
```

- The λ-expression C encoding `searchFrom` satisfies the following property:

$$
\begin{aligned}
C\,[n][n_1]\cdots[n_k] \quad \xrightarrow{*}_{\beta} \quad & \mathit{if}([\mathit{iszero}]([g][n][n_1]\cdots[n_k])) \\
& \mathit{then}\,[n] \\
& \mathit{else}\,(C([\mathit{succ}][n])[n_1]\cdots[n_k])
\end{aligned}
$$

Recursive definitions

- Suppose C satisfies the following property:

$$
\begin{aligned}
C \quad \xrightarrow{*}_{\beta} \quad & (\lambda c\, y x_1 \cdots x_k .\, \mathit{if}([\mathit{iszero}]([g]\, y x_1 \cdots x_k)) \\
& \qquad \mathit{then}\ y\ \mathit{else}\, (c([\mathit{succ}]\, y) x_1 \cdots x_k)) \quad C
\end{aligned}
$$

Recursive definitions

- Suppose C satisfies the following property:

$$\begin{aligned} C \quad \xrightarrow{*}_{\beta} \quad & (\lambda c\, y x_1 \cdots x_k.\ \mathit{if}\,([\mathit{iszero}]([g]\, y x_1 \cdots x_k)) \\ & \qquad \mathit{then}\ y\ \mathit{else}\,(c([\mathit{succ}]\, y) x_1 \cdots x_k))\ \ C \end{aligned}$$

- Then it satisfies the following:

$$\begin{aligned} C\,[n][n_1]\cdots[n_k] \quad \xrightarrow{*}_{\beta} \quad & \mathit{if}\,([\mathit{iszero}]([g][n][n_1]\cdots[n_k])) \\ & \mathit{then}\,[n] \\ & \mathit{else}\,(C([\mathit{succ}][n])[n_1]\cdots[n_k]) \end{aligned}$$

Recursive definitions

- Suppose C satisfies the following property:

$$\begin{aligned} C \quad \xrightarrow{*}_{\beta} \quad & (\lambda c\, y\, x_1 \cdots x_k.\, \mathit{if}([\mathit{iszero}]([g]\, y\, x_1 \cdots x_k)) \\ & \quad \mathit{then}\ y\ \mathit{else}\,(c([\mathit{succ}]\, y)x_1 \cdots x_k))\ \ C \end{aligned}$$

- Then it satisfies the following:

$$\begin{aligned} C\,[n][n_1]\cdots[n_k] \quad \xrightarrow{*}_{\beta} \quad & \mathit{if}([\mathit{iszero}]([g][n][n_1]\cdots[n_k])) \\ & \mathit{then}\,[n] \\ & \mathit{else}\,(C([\mathit{succ}][n])[n_1]\cdots[n_k]) \end{aligned}$$

- So letting F be

$$\begin{aligned} & \lambda c\, y\, x_1 \cdots x_k.\mathit{if}([\mathit{iszero}]([g]\, y\, x_1 \cdots x_k)) \\ & \quad \mathit{then}\ y\ \mathit{else}\,(c([\mathit{succ}]\, y)x_1 \cdots x_k) \end{aligned}$$

we want $C \xrightarrow{*}_{\beta} F\, C$

Recursive definitions and the Y combinator

- Given a λ-expression F, find an expression C such that

$$C \xrightarrow{*}_{\beta} F\ C$$

Recursive definitions and the Y combinator

- Given a λ-expression F, find an expression C such that

$$C \xrightarrow{*}_{\beta} F\ C$$

- Define $\mathbf{Y} = \lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$

Recursive definitions and the Y combinator

- Given a λ-expression F, find an expression C such that

$$C \xrightarrow{*}_{\beta} F\ C$$

- Define $\mathbf{Y} = \lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$
- $\mathbf{Y}\ F \longrightarrow_{\beta} (\lambda x.F(xx))(\lambda x.F(xx)) \longrightarrow_{\beta} F(\lambda x.F(xx))(\lambda x.F(xx))$

Recursive definitions and the $\mathbf{Y}$ combinator

- Given a λ-expression F, find an expression C such that

$$C \xrightarrow{*}_{\beta} F\ C$$

- Define $\mathbf{Y} = \lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$
- $\mathbf{Y}\ F \longrightarrow_{\beta} (\lambda x.F(xx))(\lambda x.F(xx)) \longrightarrow_{\beta} F(\lambda x.F(xx))(\lambda x.F(xx))$
- $F(\mathbf{Y}\ F) \longrightarrow_{\beta} F(\lambda x.F(xx))(\lambda x.F(xx))$

Recursive definitions and the $\mathbf{Y}$ combinator

- Given a λ-expression F, find an expression C such that

$$C \xrightarrow{*}_{\beta} F\ C$$

- Define $\mathbf{Y} = \lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$
- $\mathbf{Y}\ F \longrightarrow_{\beta} (\lambda x.F(xx))(\lambda x.F(xx)) \longrightarrow_{\beta} F(\lambda x.F(xx))(\lambda x.F(xx))$
- $F(\mathbf{Y}\ F) \longrightarrow_{\beta} F(\lambda x.F(xx))(\lambda x.F(xx))$
- So there is a G such that $\mathbf{Y}\ F \xrightarrow{*}_{\beta} G$ and $F(\mathbf{Y}\ F) \xrightarrow{*}_{\beta} G$

Recursive definitions and the Y combinator

- Given a λ-expression F, find an expression C such that

$$C \xrightarrow{*}_{\beta} F\ C$$

- Define $\mathbf{Y} = \lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$
- $\mathbf{Y}\ F \longrightarrow_{\beta} (\lambda x.F(xx))(\lambda x.F(xx)) \longrightarrow_{\beta} F(\lambda x.F(xx))(\lambda x.F(xx))$
- $F(\mathbf{Y}\ F) \longrightarrow_{\beta} F(\lambda x.F(xx))(\lambda x.F(xx))$
- So there is a G such that $\mathbf{Y}\ F \xrightarrow{*}_{\beta} G$ and $F(\mathbf{Y}\ F) \xrightarrow{*}_{\beta} G$
- We say that $\mathbf{Y}\ F =_{\beta} F\ (\mathbf{Y}\ F)$

Recursive definitions and the $\mathbf{Y}$ combinator

- Given a λ-expression F, find an expression C such that

$$C \xrightarrow{*}_{\beta} F\ C$$

- Define $\mathbf{Y} = \lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$
- $\mathbf{Y}\ F \longrightarrow_{\beta} (\lambda x.F(xx))(\lambda x.F(xx)) \longrightarrow_{\beta} F(\lambda x.F(xx))(\lambda x.F(xx))$
- $F(\mathbf{Y}\ F) \longrightarrow_{\beta} F(\lambda x.F(xx))(\lambda x.F(xx))$
- So there is a G such that $\mathbf{Y}\ F \xrightarrow{*}_{\beta} G$ and $F(\mathbf{Y}\ F) \xrightarrow{*}_{\beta} G$
- We say that $\mathbf{Y}\ F =_{\beta} F\ (\mathbf{Y}\ F)$
- For any F, $\mathbf{Y}\ F$ is a C such that $C =_{\beta} F\ C$

Recursive definitions and the Θ combinator

- Given a λ-expression F, find an expression C such that

$$C \xrightarrow{*}_{\beta} F\ C$$

Recursive definitions and the Θ combinator

- Given a λ-expression F, find an expression C such that

$$C \xrightarrow{*}_{\beta} F\ C$$

- Define $\Theta = (\lambda x\, y.\, y(x\, x\, y))(\lambda x\, y.\, y(x\, x\, y))$

Recursive definitions and the Θ combinator

- Given a λ-expression F, find an expression C such that

$$C \xrightarrow{*}_{\beta} F\ C$$

- Define $\Theta = (\lambda x\,y.y(x\,x\,y))(\lambda x\,y.y(x\,x\,y))$
- $\Theta\ F = (\lambda x\,y.y(x\,x\,y))(\lambda x\,y.y(x\,x\,y))\ F \longrightarrow_{\beta}$
  $(\ \lambda y.y(\ (\lambda x\,y.y(x\,x\,y))\,(\lambda x\,y.y(x\,x\,y))\ y)\ )\ F \longrightarrow_{\beta}$
  $F\ (\ (\lambda x\,y.y(x\,x\,y))\,(\lambda x\,y.y(x\,x\,y))\ F\ ) = F\ (\Theta\ F\ )$

Recursive definitions and the Θ combinator

- Given a λ-expression F, find an expression C such that

$$C \xrightarrow{*}_{\beta} F\ C$$

- Define $\Theta = (\lambda x\,y.\,y(x\,x\,y))(\lambda x\,y.\,y(x\,x\,y))$
- $\Theta\ F = (\lambda x\,y.\,y(x\,x\,y))(\lambda x\,y.\,y(x\,x\,y))\ F \longrightarrow_{\beta}$
  $(\ \lambda y.\,y(\ (\lambda x\,y.\,y(x\,x\,y))\,(\lambda x\,y.\,y(x\,x\,y))\ y)\ )\ F \longrightarrow_{\beta}$
  $F\ (\ (\lambda x\,y.\,y(x\,x\,y))\,(\lambda x\,y.\,y(x\,x\,y))\ F\,) = F\ (\Theta\ F\,)$
- Thus $\Theta\ F \xrightarrow{*}_{\beta} F\ (\Theta\ F\,)$

Recursive definitions and the Θ combinator

- Given a λ-expression F, find an expression C such that

$$C \xrightarrow{*}_{\beta} F\ C$$

- Define $\Theta = (\lambda x\, y. y(x\, x\, y))(\lambda x\, y. y(x\, x\, y))$
- $\Theta\ F = (\lambda x\, y. y(x\, x\, y))(\lambda x\, y. y(x\, x\, y))\ F \longrightarrow_{\beta}$
  $(\ \lambda y. y(\ (\lambda x\, y. y(x\, x\, y))\ (\lambda x\, y. y(x\, x\, y))\ y)\ )\ F \longrightarrow_{\beta}$
  $F\ (\ (\lambda x\, y. y(x\, x\, y))\ (\lambda x\, y. y(x\, x\, y))\ F) = F\ (\Theta\ F)$
- Thus $\Theta\ F \xrightarrow{*}_{\beta} F\ (\Theta\ F)$
- For any F, $\Theta\ F$ is a C such that $C \xrightarrow{*}_{\beta} F\ C$

Recursive definitions and the Θ combinator

- Given a λ-expression F, find an expression C such that

$$C \xrightarrow{*}_{\beta} F\ C$$

- Define $\Theta = (\lambda x\,y.y(x\,x\,y))(\lambda x\,y.y(x\,x\,y))$
- $\Theta\ F = (\lambda x\,y.y(x\,x\,y))(\lambda x\,y.y(x\,x\,y))\ F \longrightarrow_{\beta}$
  $(\ \lambda y.y(\ (\lambda x\,y.y(x\,x\,y))\ (\lambda x\,y.y(x\,x\,y))\ y)\ )\ F \longrightarrow_{\beta}$
  $F\ (\ (\lambda x\,y.y(x\,x\,y))\ (\lambda x\,y.y(x\,x\,y))\ F\ ) = F\ (\Theta\ F)$
- Thus $\Theta\ F \xrightarrow{*}_{\beta} F\ (\Theta\ F)$
- For any F, $\Theta\ F$ is a C such that $C \xrightarrow{*}_{\beta} F\ C$
- $\mathbf{Y}$ and Θ are **fixed-point combinators**