Programming Language Concepts: Lecture 16

S P Suresh

Chennai Mathematical Institute

spsuresh@cmi.ac.in

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

March 30, 2016

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

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

Encoding primitive recursion

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

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

Encoding μ-recursion

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

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

Encoding μ-recursion

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

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

- Implement the `while` loop using recursion:

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

Encoding μ-recursion

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

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

- Implement the `while` loop using recursion:

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

- Need ways to encode booleans, if-then-else, test for zero and recursive definitions in λ-calculus

Encoding booleans

- $[true] = \lambda xy.x$

Encoding booleans

- $[true] = \lambda xy.x$
- $[false] = \lambda xy.y$

Encoding booleans

- $[\mathit{true}] = \lambda xy.x$
- $[\mathit{false}] = \lambda xy.y$
- $[\mathit{if\text{-}then\text{-}else}] = \lambda bxy.bxy$

Encoding booleans

- $[true] = \lambda xy.x$
- $[false] = \lambda xy.y$
- $[\textit{if-then-else}] = \lambda bxy.bxy$
- Syntactic sugar: $[\textit{if-then-else}]\, b\, f\, g$ is written as $\textit{if}\ b\ \textit{then}\ f\ \textit{else}\ g$

Encoding booleans

- $[true] = \lambda xy.x$
- $[false] = \lambda xy.y$
- $[\mathit{if\text{-}then\text{-}else}] = \lambda bxy.bxy$
- Syntactic sugar: $[\mathit{if\text{-}then\text{-}else}]\, b\, f\, g$ is written as $\mathit{if}\ b\ \mathit{then}\ f\ \mathit{else}\ g$

$$\begin{aligned}
\mathit{if}\,[true]\ \mathit{then}\ f\ \mathit{else}\ g & = \\
(\lambda bxy.bxy)(\lambda xy.x) f\, g & \longrightarrow_\beta (\lambda xy.(\lambda xy.x)xy) f\, g \\
& \longrightarrow_\beta (\lambda y.(\lambda xy.x) f\, y) g \\
& \longrightarrow_\beta (\lambda xy.x) f\, g \\
& \longrightarrow_\beta (\lambda y.f) g \\
& \longrightarrow_\beta f
\end{aligned}$$

- $[\mathit{true}] = \lambda xy.x$
- $[\mathit{false}] = \lambda xy.y$
- $[\mathit{if\text{-}then\text{-}else}] = \lambda bxy.bxy$
- Syntactic sugar: $[\mathit{if\text{-}then\text{-}else}]\, b\, f\, g$ is written as $\mathit{if}\ b\ \mathit{then}\ f\ \mathit{else}\ g$

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

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

Encoding test for zero

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

Encoding test for zero

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

Encoding test for zero

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

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

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

Encoding test for zero

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

Encoding μ-recursion

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

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

Encoding μ-recursion

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

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

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

Encoding μ-recursion

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

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

Encoding μ-recursion

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

```
f(n1, n2, ..., nk) = check(0, n1, n2, ..., nk)
    where
        check(n, n1, n2, ..., nk) {
            if (iszero(g(n, n1, n2, ..., nk))) {return n;}
            else return check(n+1, 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:

```
f(n1, n2, ..., nk) = check(0, n1, n2, ..., nk)
    where
        check(n, n1, n2, ..., nk) {
            if (iszero(g(n, n1, n2, ..., nk))) {return n;}
            else return check(n+1, 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
- $[f] = \lambda x_1 \cdots x_k.W\,[0]\,W$
- $[f][n_1]\cdots[n_k] \xrightarrow{*}_{\beta} W'\,[0]\,W'$

Encoding μ-recursion

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

```
f(n1, n2, ..., nk) = check(0, n1, n2, ..., nk)
    where
        check(n, n1, n2, ..., nk) {
            if (iszero(g(n, n1, n2, ..., nk))) {return n;}
            else return check(n+1, 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
- $[f] = \lambda x_1 \cdots x_k.W\,[0]\,W$
- $[f][n_1]\cdots[n_k] \xrightarrow{*}_{\beta} W'\,[0]\,W'$
  - $W' = W[x_1 := [n_1], \ldots, x_k := [n_k]]$

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

Encoding μ-recursion

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

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

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 $[g][i][n_1]\cdots[n_k] \xrightarrow{*}_{\beta} [0]$

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 $[iszero]([g][i][n_1]\cdots[n_k]) \xrightarrow{*}_\beta [iszero][0] \xrightarrow{*}_\beta [true]$

- 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 $[iszero]([g][i][n_1] \cdots [n_k]) \xrightarrow{*}_\beta [iszero][0] \xrightarrow{*}_\beta [true]$
  - So

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

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 $[iszero]([g][i][n_1]\cdots[n_k]) \xrightarrow{*}_\beta [iszero][0] \xrightarrow{*}_\beta [false]$

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

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

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

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 $g(n,\vec{n}) = 0$ and $g(m,\vec{n}) > 0$ for all $m < n$
- $W' [0] W' \xrightarrow{*}_\beta W' [1] W' \xrightarrow{*}_\beta W' [2] W' \xrightarrow{*}_\beta \ldots \xrightarrow{*}_\beta W' [n] W' \xrightarrow{*}_\beta [n]$

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 $g(n,\vec{n}) = 0$ and $g(m,\vec{n}) > 0$ for all $m < n$
- $W'[0]W' \xrightarrow{*}_{\beta} W'[1]W' \xrightarrow{*}_{\beta} W'[2]W' \xrightarrow{*}_{\beta} \ldots \xrightarrow{*}_{\beta} W'[n]W' \xrightarrow{*}_{\beta} [n]$
- Thus $[f][n_1]\cdots[n_k] \xrightarrow{*}_{\beta} W'[n]W' \xrightarrow{*}_{\beta} [n] = [\mu n.g(n,\vec{n}) = 0]$

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

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

Recursive definitions

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

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

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

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

Recursive definitions

- Suppose C satisfies the following property:

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

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:

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

- Suppose C satisfies the following property:

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

- Then it satisfies the following:

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

- So letting F be

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

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 **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 **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 **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 $\Theta = (\lambda xy.y(xxy))(\lambda xy.y(xxy))$

Recursive definitions and the Θ combinator

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

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

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

Recursive definitions and the Θ combinator

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

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

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