CS 4110

Programming Languages & Logics

Lecture 16
Programming in the λ-calculus

Review: Church Booleans

We can encode TRUE, FALSE, and IF, as:

$$\text{TRUE} \triangleq \lambda x.\, \lambda y.\, x$$
$$\text{FALSE} \triangleq \lambda x.\, \lambda y.\, y$$
$$\text{IF} \triangleq \lambda b.\, \lambda t.\, \lambda f.\, b\ t\ f$$

This way, IF behaves how it ought to:

$$\text{IF TRUE}\ v_t\ v_f \rightarrow^* v_t$$
$$\text{IF FALSE}\ v_t\ v_f \rightarrow^* v_f$$

Review: Church Numerals

Church numerals encode a number n as a function that takes f and x, and applies f to x n times.

$$\begin{aligned}
\overline{0} &\triangleq \lambda f.\, \lambda x.\, x \\
\overline{1} &\triangleq \lambda f.\, \lambda x.\, f\, x \\
\overline{2} &\triangleq \lambda f.\, \lambda x.\, f\, (f\, x)
\end{aligned}$$

We can define other functions on integers:

$$\mathrm{SUCC} \triangleq \lambda n.\, \lambda f.\, \lambda x.\, f\, (n\, f\, x)$$

Review: Church Numerals

Church numerals encode a number n as a function that takes f and x, and applies f to x n times.

$$\begin{aligned} \overline{0} &\triangleq \lambda f.\, \lambda x.\, x \\ \overline{1} &\triangleq \lambda f.\, \lambda x.\, f\,x \\ \overline{2} &\triangleq \lambda f.\, \lambda x.\, f\,(f\,x) \end{aligned}$$

We can define other functions on integers:

$$\begin{aligned} \text{SUCC} &\triangleq \lambda n.\, \lambda f.\, \lambda x.\, f\,(n\,f\,x) \\ \text{PLUS} &\triangleq \lambda n_1.\, \lambda n_2.\, n_1\ \text{SUCC}\ n_2 \end{aligned}$$

Review: Church Numerals

Church numerals encode a number n as a function that takes f and x, and applies f to x n times.

$$\begin{array}{rcl}\overline{0} & \triangleq & \lambda f.\, \lambda x.\, x \\ \overline{1} & \triangleq & \lambda f.\, \lambda x.\, f\, x \\ \overline{2} & \triangleq & \lambda f.\, \lambda x.\, f\,(f\, x)\end{array}$$

We can define other functions on integers:

$$\begin{array}{rcl}\mathrm{SUCC} & \triangleq & \lambda n.\, \lambda f.\, \lambda x.\, f\,(n\, f\, x) \\ \mathrm{PLUS} & \triangleq & \lambda n_1.\, \lambda n_2.\, n_1\; \mathrm{SUCC}\; n_2 \\ \mathrm{TIMES} & \triangleq & \lambda n_1.\, \lambda n_2.\, n_1\; (\mathrm{PLUS}\; \mathrm{n}_2)\; \overline{0}\end{array}$$

Review: Church Numerals

Church numerals encode a number *n* as a function that takes *f* and *x*, and applies *f* to *x* *n* times.

$\Omega = (\lambda x.\, x\, x)\, (\lambda x.\, x\, x)$

$$
\begin{aligned}
\overline{0} &\triangleq \lambda f.\, \lambda x.\, x \\
\overline{1} &\triangleq \lambda f.\, \lambda x.\, f\, x \\
\overline{2} &\triangleq \lambda f.\, \lambda x.\, f\, (f\, x)
\end{aligned}
$$

We can define other functions on integers:

$$
\begin{aligned}
\text{SUCC} &\triangleq \lambda n.\, \lambda f.\, \lambda x.\, f\, (n\, f\, x) \\
\text{PLUS} &\triangleq \lambda n_1.\, \lambda n_2.\, n_1\ \text{SUCC}\ n_2 \\
\text{TIMES} &\triangleq \lambda n_1.\, \lambda n_2.\, n_1\ (\text{PLUS}\ \mathrm{n}_2)\ \overline{0} \\
\text{ISZERO} &\triangleq \lambda n.\, n\ (\lambda z.\, \text{FALSE})\ \text{TRUE}
\end{aligned}
$$

Recursive Functions

How would we write recursive functions like factorial?

Recursive Functions

How would we write recursive functions like factorial?

We'd like to write it like this...

$$\text{FACT} \triangleq \lambda n.\ \text{IF}\ (\text{ISZERO}\ n)\ 1\ (\text{TIMES}\ n\ (\text{FACT}\ (\text{PRED}\ n)))$$

Recursive Functions

How would we write recursive functions like factorial?

We'd like to write it like this...

$$\mathrm{FACT} \triangleq \lambda n.\, \mathrm{IF}\ (\mathrm{ISZERO}\ n)\ 1\ (\mathrm{TIMES}\ n\ (\mathrm{FACT}\ (\mathrm{PRED}\ n)))$$

In slightly more readable notation this is...

$$\mathrm{FACT} \triangleq \lambda n.\, \textbf{if}\ n = 0\ \textbf{then}\ 1\ \textbf{else}\ n \times \mathrm{FACT}\ (n-1)$$

...but this is an equation, not a definition!

Recursion removal trick

We can perform a “trick” to define a function FACT that satisfies the recursive equation on the previous slide.

Recursion removal trick

We can perform a "trick" to define a function FACT that satisfies the recursive equation on the previous slide.

Define a new function FACT$'$ that takes a function f as an argument. Then, for "recursive" calls, it uses $f\,f$:

$$\text{FACT}' \triangleq \lambda f.\, \lambda n.\, \textbf{if } n = 0 \textbf{ then } 1 \textbf{ else } n \times ((f\,f)\,(n-1))$$

Recursion removal trick

We can perform a “trick” to define a function FACT that satisfies the recursive equation on the previous slide.

Define a new function FACT′ that takes a function *f* as an argument. Then, for “recursive” calls, it uses *f f*:

$$\mathsf{FACT}' \triangleq \lambda f.\, \lambda n.\, \textbf{if } n = 0 \textbf{ then } 1 \textbf{ else } n \times ((f\, f)\, (n-1))$$

Then define FACT as FACT′ applied to itself:

$$\mathsf{FACT} \triangleq \mathsf{FACT}'\ \mathsf{FACT}'$$

Example

Let's try evaluating FACT on 3...

FACT 3

Example

Let's try evaluating FACT on 3...

$\text{FACT } 3 = (\text{FACT}' \text{ FACT}')\ 3$

Example

Let's try evaluating FACT on 3...

$$
\begin{aligned}
\text{FACT } 3 &= (\text{FACT}' \ \text{FACT}') \ 3 \\
&= ((\lambda f.\ \lambda n.\ \textbf{if } n = 0 \textbf{ then } 1 \textbf{ else } n \times ((f\, f)\ (n-1)))\ \text{FACT}')\ 3
\end{aligned}
$$

Example

Let's try evaluating FACT on 3...

$$
\begin{aligned}
\mathrm{FACT}\;3 &= (\mathrm{FACT}'\;\mathrm{FACT}')\;3 \\
&= ((\lambda f.\,\lambda n.\,\textbf{if}\;n = 0\;\textbf{then}\;1\;\textbf{else}\;n \times ((f\,f)\;(n-1)))\;\mathrm{FACT}')\;3 \\
&\to (\lambda n.\,\textbf{if}\;n = 0\;\textbf{then}\;1\;\textbf{else}\;n \times ((\mathrm{FACT}'\;\mathrm{FACT}')\;(n-1)))\;3
\end{aligned}
$$

Example

Let's try evaluating FACT on 3...

$$
\begin{aligned}
\text{FACT}\ 3 &= (\text{FACT}'\ \text{FACT}')\ 3 \\
&= ((\lambda f.\ \lambda n.\ \textbf{if}\ n = 0\ \textbf{then}\ 1\ \textbf{else}\ n \times ((f\ f)\ (n-1)))\ \text{FACT}')\ 3 \\
&\to (\lambda n.\ \textbf{if}\ n = 0\ \textbf{then}\ 1\ \textbf{else}\ n \times ((\text{FACT}'\ \text{FACT}')\ (n-1)))\ 3 \\
&\to \textbf{if}\ 3 = 0\ \textbf{then}\ 1\ \textbf{else}\ 3 \times ((\text{FACT}'\ \text{FACT}')\ (3-1))
\end{aligned}
$$

Example

Let's try evaluating FACT on 3...

$$
\begin{aligned}
\mathsf{FACT}\ 3 &= (\mathsf{FACT}'\ \mathsf{FACT}')\ 3 \\
&= ((\lambda f.\ \lambda n.\ \textbf{if}\ n = 0\ \textbf{then}\ 1\ \textbf{else}\ n \times ((f\,f)\ (n-1)))\ \mathsf{FACT}')\ 3 \\
&\to (\lambda n.\ \textbf{if}\ n = 0\ \textbf{then}\ 1\ \textbf{else}\ n \times ((\mathsf{FACT}'\ \mathsf{FACT}')\ (n-1)))\ 3 \\
&\to \textbf{if}\ 3 = 0\ \textbf{then}\ 1\ \textbf{else}\ 3 \times ((\mathsf{FACT}'\ \mathsf{FACT}')\ (3-1)) \\
&\to 3 \times ((\mathsf{FACT}'\ \mathsf{FACT}')\ (3-1))
\end{aligned}
$$

Example

Let's try evaluating FACT on 3...

$$
\begin{aligned}
\mathrm{FACT}\ 3 &= (\mathrm{FACT}'\ \mathrm{FACT}')\ 3 \\
&= ((\lambda f.\ \lambda n.\ \textbf{if}\ n = 0\ \textbf{then}\ 1\ \textbf{else}\ n \times ((f\ f)\ (n-1)))\ \mathrm{FACT}')\ 3 \\
&\to (\lambda n.\ \textbf{if}\ n = 0\ \textbf{then}\ 1\ \textbf{else}\ n \times ((\mathrm{FACT}'\ \mathrm{FACT}')\ (n-1)))\ 3 \\
&\to \textbf{if}\ 3 = 0\ \textbf{then}\ 1\ \textbf{else}\ 3 \times ((\mathrm{FACT}'\ \mathrm{FACT}')\ (3-1)) \\
&\to 3 \times ((\mathrm{FACT}'\ \mathrm{FACT}')\ (3-1)) \\
&= 3 \times (\mathrm{FACT}\ (3-1))
\end{aligned}
$$

Example

Let's try evaluating FACT on 3...

$$\begin{aligned}
\text{FACT}\ 3 &= (\text{FACT}'\ \text{FACT}')\ 3 \\
&= ((\lambda f.\, \lambda n.\, \textbf{if}\ n = 0\ \textbf{then}\ 1\ \textbf{else}\ n \times ((f\, f)\ (n-1)))\ \text{FACT}')\ 3 \\
&\to (\lambda n.\, \textbf{if}\ n = 0\ \textbf{then}\ 1\ \textbf{else}\ n \times ((\text{FACT}'\ \text{FACT}')\ (n-1)))\ 3 \\
&\to \textbf{if}\ 3 = 0\ \textbf{then}\ 1\ \textbf{else}\ 3 \times ((\text{FACT}'\ \text{FACT}')\ (3-1)) \\
&\to 3 \times ((\text{FACT}'\ \text{FACT}')\ (3-1)) \\
&= 3 \times (\text{FACT}\ (3-1)) \\
&\to \ldots \\
&\to 3 \times 2 \times 1 \times 1
\end{aligned}$$

Example

Let's try evaluating FACT on 3...

$$
\begin{aligned}
\mathrm{FACT}\ 3 &= (\mathrm{FACT}'\ \mathrm{FACT}')\ 3 \\
&= ((\lambda f.\ \lambda n.\ \textbf{if}\ n = 0\ \textbf{then}\ 1\ \textbf{else}\ n \times ((f\,f)\ (n-1)))\ \mathrm{FACT}')\ 3 \\
&\rightarrow (\lambda n.\ \textbf{if}\ n = 0\ \textbf{then}\ 1\ \textbf{else}\ n \times ((\mathrm{FACT}'\ \mathrm{FACT}')\ (n-1)))\ 3 \\
&\rightarrow \textbf{if}\ 3 = 0\ \textbf{then}\ 1\ \textbf{else}\ 3 \times ((\mathrm{FACT}'\ \mathrm{FACT}')\ (3-1)) \\
&\rightarrow 3 \times ((\mathrm{FACT}'\ \mathrm{FACT}')\ (3-1)) \\
&= 3 \times (\mathrm{FACT}\ (3-1)) \\
&\rightarrow \ldots \\
&\rightarrow 3 \times 2 \times 1 \times 1 \\
&\rightarrow^{*} 6
\end{aligned}
$$

Fixed point combinators

Our "trick" requires following human-readable instructions. Write a different function f' that takes itself as an argument and uses self-application for recursive calls, and then define f as $f'\ f'$.

Fixed point combinators

Our “trick” requires following human-readable instructions. Write a different function f' that takes itself as an argument and uses self-application for recursive calls, and then define f as $f'\ f'$.

There is another way: fixed points!

Fixed point combinators

Our "trick" requires following human-readable instructions. Write a different function f' that takes itself as an argument and uses self-application for recursive calls, and then define f as $f'\ f'$.

There is another way: fixed points!

Consider factorial again. It is a fixed point of the following:

$$G \triangleq \lambda f.\, \lambda n.\, \textbf{if}\ n = 0\ \textbf{then}\ 1\ \textbf{else}\ n \times (f\,(n-1))$$

Fixed point combinators

Our “trick” requires following human-readable instructions. Write a different function f' that takes itself as an argument and uses self-application for recursive calls, and then define f as $f'\ f'$.

There is another way: fixed points!

Consider factorial again. It is a fixed point of the following:

$$G \triangleq \lambda f.\, \lambda n.\, \textbf{if}\ n = 0\ \textbf{then}\ 1\ \textbf{else}\ n \times (f\,(n-1))$$

Recall that if g if a fixed point of G, then $G\,g = g$. To see that any fixed point g is a real factorial function, try evaluating it:

$$g\,5 = (G\,g)\,5$$

Fixed point combinators

Our “trick” requires following human-readable instructions. Write a different function f' that takes itself as an argument and uses self-application for recursive calls, and then define f as $f'\ f'$.

There is another way: fixed points!

Consider factorial again. It is a fixed point of the following:

$$G \triangleq \lambda f.\, \lambda n.\, \textbf{if } n = 0 \textbf{ then } 1 \textbf{ else } n \times (f\,(n - 1))$$

Recall that if g if a fixed point of G, then $G\,g = g$. To see that any fixed point g is a real factorial function, try evaluating it:

$$\begin{aligned} g\,5 &= (G\,g)\,5 \\ &\to^* 5 \times (g\,4) \end{aligned}$$

Fixed point combinators

Our “trick” requires following human-readable instructions. Write a different function f' that takes itself as an argument and uses self-application for recursive calls, and then define f as $f'\ f'$.

There is another way: fixed points!

Consider factorial again. It is a fixed point of the following:

$$G \triangleq \lambda f.\, \lambda n.\, \textbf{if}\ n = 0\ \textbf{then}\ 1\ \textbf{else}\ n \times (f\,(n-1))$$

Recall that if g if a fixed point of G, then $G\,g = g$. To see that any fixed point g is a real factorial function, try evaluating it:

$$\begin{aligned} g\,5 &= (G\,g)\,5 \\ &\to^* 5 \times (g\,4) \\ &= 5 \times ((G\,g)\,4) \end{aligned}$$

Fixed point combinators

How can we generate the fixed point of G?

In denotational semantics, finding fixed points took a lot of math. In the λ-calculus, we just need a suitable combinator...

Y Combinator

The (infamous) Y combinator is defined as

$$\mathsf{Y} \triangleq \lambda f.\,(\lambda x.\,f\,(x\,x))\,(\lambda x.\,f\,(x\,x))$$

We say that Y is a *fixed point combinator* because Y f is a fixed point of f (for any lambda term f).

Y Combinator

The (infamous) Y combinator is defined as

$$\mathsf{Y} \triangleq \lambda f.\, (\lambda x.\, f\,(x\,x))\,(\lambda x.\, f\,(x\,x))$$

We say that Y is a *fixed point combinator* because Y f is a fixed point of f (for any lambda term f).

What happens when we evaluate Y G under CBV?

$$Y\,G = \big((\lambda f.\, (\lambda x.\, f\,(x\,x))\,(\lambda x.\, f\,(x\,x)))\;(\lambda f.\ \text{if}\ \ldots\ldots)\big)$$

$$\rightarrow (\lambda x.\, G\,(x\,x))\,(\lambda x.\, G\,(x\,x))$$

$$\rightarrow G\,((\lambda x.\, G\,(x\,x))\,(\lambda x.\, G\,(x\,x)))$$

$$\rightarrow G\,(G\,((\lambda x.\, G\,(x\,x))\,(\lambda x.\, G\,(x\,x))))$$

Z Combinator

To avoid this issue, we'll use a slight variant of the Y combinator, called Z, which is easier to use under CBV.

Z Combinator

To avoid this issue, we'll use a slight variant of the Y combinator, called Z, which is easier to use under CBV.

$$\mathsf{Z} \triangleq \lambda f.\, (\lambda x.\, f\, (\lambda y.\, x\, x\, y))\, (\lambda x.\, f\, (\lambda y.\, x\, x\, y))$$

$\lambda y.\ e\ y$

Example

Let’s see Z in action, on our function G.

FACT

Example

Let's see Z in action, on our function G.

$$
\begin{array}{rl}
& \mathrm{FACT} \\
= & \mathrm{Z}\, G
\end{array}
$$

Example

Let's see Z in action, on our function G.

$$
\begin{array}{ll}
 & \mathrm{FACT} \\
= & \mathrm{Z}\, G \\
= & (\lambda f.\, (\lambda x.\, f\, (\lambda y.\, x\, x\, y))\, (\lambda x.\, f\, (\lambda y.\, x\, x\, y)))\, G
\end{array}
$$

Example

Let's see Z in action, on our function G.

$$
\begin{array}{ll}
 & \mathrm{FACT} \\
= & \mathrm{Z}\, G \\
= & (\lambda f.\, (\lambda x.\, f\, (\lambda y.\, x\, x\, y))\, (\lambda x.\, f\, (\lambda y.\, x\, x\, y)))\, G \\
\to & (\lambda x.\, G\, (\lambda y.\, x\, x\, y))\, (\lambda x.\, G\, (\lambda y.\, x\, x\, y))
\end{array}
$$

Example

Let's see Z in action, on our function G.

$$
\begin{array}{ll}
 & \mathsf{FACT} \\
= & \mathsf{Z}\, G \\
= & (\lambda f.\, (\lambda x.\, f\, (\lambda y.\, x\, x\, y))\, (\lambda x.\, f\, (\lambda y.\, x\, x\, y)))\, G \\
\to & (\lambda x.\, G\, (\lambda y.\, x\, x\, y))\, (\lambda x.\, G\, (\lambda y.\, x\, x\, y)) \\
\to & G\, (\lambda y.\, (\lambda x.\, G\, (\lambda y.\, x\, x\, y))\, (\lambda x.\, G\, (\lambda y.\, x\, x\, y))\, y)
\end{array}
$$

Example

Let's see Z in action, on our function G.

$$
\begin{array}{ll}
 & \mathrm{FACT} \\
= & \mathrm{Z}\, G \\
= & (\lambda f.\, (\lambda x.\, f\, (\lambda y.\, x\, x\, y))\, (\lambda x.\, f\, (\lambda y.\, x\, x\, y)))\, G \\
\to & (\lambda x.\, G\, (\lambda y.\, x\, x\, y))\, (\lambda x.\, G\, (\lambda y.\, x\, x\, y)) \\
\to & G\, (\lambda y.\, (\lambda x.\, G\, (\lambda y.\, x\, x\, y))\, (\lambda x.\, G\, (\lambda y.\, x\, x\, y))\, y) \\
= & (\lambda f.\, \lambda n.\, \mathbf{if}\ n = 0\ \mathbf{then}\ 1\ \mathbf{else}\ n \times (f\, (n-1))) \\
 & \qquad (\lambda y.\, (\lambda x.\, G\, (\lambda y.\, x\, x\, y))\, (\lambda x.\, G\, (\lambda y.\, x\, x\, y))\, y)
\end{array}
$$

Example

Let's see Z in action, on our function G.

$$
\begin{array}{ll}
 & \mathrm{FACT} \\
= & \mathrm{Z}\, G \\
= & (\lambda f.\, (\lambda x.\, f\,(\lambda y.\, x\,x\,y))\,(\lambda x.\, f\,(\lambda y.\, x\,x\,y)))\; G \\
\to & (\lambda x.\, G\,(\lambda y.\, x\,x\,y))\,(\lambda x.\, G\,(\lambda y.\, x\,x\,y)) \\
\to & G\,(\lambda y.\, (\lambda x.\, G\,(\lambda y.\, x\,x\,y))\,(\lambda x.\, G\,(\lambda y.\, x\,x\,y))\,y) \\
= & (\lambda f.\, \lambda n.\, \textbf{if}\ n = 0\ \textbf{then}\ 1\ \textbf{else}\ n \times (f\,(n-1))) \\
 & \qquad (\lambda y.\, (\lambda x.\, G\,(\lambda y.\, x\,x\,y))\,(\lambda x.\, G\,(\lambda y.\, x\,x\,y))\,y) \\
\to & \lambda n.\, \textbf{if}\ n = 0\ \textbf{then}\ 1 \\
 & \quad \textbf{else}\ n \times ((\lambda y.\, (\lambda x.\, G\,(\lambda y.\, x\,x\,y))\,(\lambda x.\, G\,(\lambda y.\, x\,x\,y))\,y)\,(n-1))
\end{array}
$$

Example

Let's see Z in action, on our function G.

$$
\begin{array}{rl}
 & \text{FACT} \\
= & \mathsf{Z}\, G \\
= & (\lambda f.\, (\lambda x.\, f\, (\lambda y.\, x\, x\, y))\, (\lambda x.\, f\, (\lambda y.\, x\, x\, y)))\, G \\
\to & (\lambda x.\, G\, (\lambda y.\, x\, x\, y))\, (\lambda x.\, G\, (\lambda y.\, x\, x\, y)) \\
\to & G\, (\lambda y.\, (\lambda x.\, G\, (\lambda y.\, x\, x\, y))\, (\lambda x.\, G\, (\lambda y.\, x\, x\, y))\, y) \\
= & (\lambda f.\, \lambda n.\, \textbf{if } n = 0 \textbf{ then } 1 \textbf{ else } n \times (f\, (n-1))) \\
 & \qquad (\lambda y.\, (\lambda x.\, G\, (\lambda y.\, x\, x\, y))\, (\lambda x.\, G\, (\lambda y.\, x\, x\, y))\, y) \\
\to & \lambda n.\, \textbf{if } n = 0 \textbf{ then } 1 \\
 & \quad \textbf{else } n \times ((\lambda y.\, (\lambda x.\, G\, (\lambda y.\, x\, x\, y))\, (\lambda x.\, G\, (\lambda y.\, x\, x\, y))\, y)\, (n-1)) \\
=_\beta & \lambda n.\, \textbf{if } n = 0 \textbf{ then } 1 \textbf{ else } n \times (\lambda y.\, (\mathsf{Z}\, G)\, y)\, (n-1)
\end{array}
$$

Example

Let's see Z in action, on our function G.

$$
\begin{aligned}
& \mathrm{FACT} \\
= \quad & \mathrm{Z}\, G \\
= \quad & (\lambda f.\, (\lambda x.\, f\, (\lambda y.\, x\, x\, y))\, (\lambda x.\, f\, (\lambda y.\, x\, x\, y)))\, G \\
\rightarrow \quad & (\lambda x.\, G\, (\lambda y.\, x\, x\, y))\, (\lambda x.\, G\, (\lambda y.\, x\, x\, y)) \\
\rightarrow \quad & G\, (\lambda y.\, (\lambda x.\, G\, (\lambda y.\, x\, x\, y))\, (\lambda x.\, G\, (\lambda y.\, x\, x\, y))\, y) \\
= \quad & (\lambda f.\, \lambda n.\, \textbf{if}\ n = 0\ \textbf{then}\ 1\ \textbf{else}\ n \times (f\, (n-1))) \\
& \qquad (\lambda y.\, (\lambda x.\, G\, (\lambda y.\, x\, x\, y))\, (\lambda x.\, G\, (\lambda y.\, x\, x\, y))\, y) \\
\rightarrow \quad & \lambda n.\, \textbf{if}\ n = 0\ \textbf{then}\ 1 \\
& \qquad \textbf{else}\ n \times ((\lambda y.\, (\lambda x.\, G\, (\lambda y.\, x\, x\, y))\, (\lambda x.\, G\, (\lambda y.\, x\, x\, y))\, y)\, (n-1)) \\
=_\beta \quad & \lambda n.\, \textbf{if}\ n = 0\ \textbf{then}\ 1\ \textbf{else}\ n \times (\lambda y.\, (\mathrm{Z}\, G)\, y)\, (n-1)) \\
=_\beta \quad & \lambda n.\, \textbf{if}\ n = 0\ \textbf{then}\ 1\ \textbf{else}\ n \times (\mathrm{Z}\, G\, (n-1))
\end{aligned}
$$

Example

Let's see Z in action, on our function G.

$$
\begin{array}{ll}
 & \mathsf{FACT} \\
= & \mathsf{Z}\, G \\
= & (\lambda f.\, (\lambda x.\, f\,(\lambda y.\, x\,x\,y))\, (\lambda x.\, f\,(\lambda y.\, x\,x\,y)))\; G \\
\rightarrow & (\lambda x.\, G\,(\lambda y.\, x\,x\,y))\, (\lambda x.\, G\,(\lambda y.\, x\,x\,y)) \\
\rightarrow & G\,(\lambda y.\, (\lambda x.\, G\,(\lambda y.\, x\,x\,y))\, (\lambda x.\, G\,(\lambda y.\, x\,x\,y))\; y) \\
= & (\lambda f.\, \lambda n.\, \textbf{if}\; n = 0\; \textbf{then}\; 1\; \textbf{else}\; n \times (f\,(n-1))) \\
 & \qquad (\lambda y.\, (\lambda x.\, G\,(\lambda y.\, x\,x\,y))\, (\lambda x.\, G\,(\lambda y.\, x\,x\,y))\; y) \\
\rightarrow & \lambda n.\, \textbf{if}\; n = 0\; \textbf{then}\; 1 \\
 & \qquad \textbf{else}\; n \times ((\lambda y.\, (\lambda x.\, G\,(\lambda y.\, x\,x\,y))\, (\lambda x.\, G\,(\lambda y.\, x\,x\,y))\; y)\; (n-1)) \\
=_\beta & \lambda n.\, \textbf{if}\; n = 0\; \textbf{then}\; 1\; \textbf{else}\; n \times (\lambda y.\, (\mathsf{Z}\, G)\; y)\; (n-1) \\
=_\beta & \lambda n.\, \textbf{if}\; n = 0\; \textbf{then}\; 1\; \textbf{else}\; n \times (\mathsf{Z}\, G\, (n-1)) \\
= & \lambda n.\, \textbf{if}\; n = 0\; \textbf{then}\; 1\; \textbf{else}\; n \times (\mathsf{FACT}\; (n-1))
\end{array}
$$

Other fixed point combinators

There are many (indeed infinitely many) fixed-point combinators. Here's a cute one:

$$\mathsf{Y}_k \triangleq (\mathsf{L\,L\,L\,L\,L\,L\,L\,L\,L\,L\,L\,L\,L\,L\,L\,L\,L\,L\,L\,L\,L\,L\,L\,L\,L\,L})$$

where

$$\begin{aligned}\mathsf{L} \triangleq\ & \lambda abcdefghijklmnopqstuvwxyzr. \\ & (r\,(t\,h\,i\,s\,i\,s\,a\,f\,i\,x\,e\,d\,p\,o\,i\,n\,t\,c\,o\,m\,b\,i\,n\,a\,t\,o\,r))\end{aligned}$$

Turing's Fixed Point Combinator

To gain some more intuition for fixed point combinators, let's derive a combinator Θ originally discovered by Turing.

Turing’s Fixed Point Combinator

To gain some more intuition for fixed point combinators, let’s derive a combinator Θ originally discovered by Turing.

We know that $\Theta\, f$ is a fixed point of f, so we have

$$\Theta\, f = f\,(\Theta\, f).$$

Turing's Fixed Point Combinator

To gain some more intuition for fixed point combinators, let's derive a combinator Θ originally discovered by Turing.

We know that $\Theta\, f$ is a fixed point of f, so we have

$$\Theta\, f = f\,(\Theta\, f).$$

$f(x) = x^2$

$f = \lambda x.\, x^2$

We can write the following recursive equation:

$$\Theta = \lambda f.\, f\,(\Theta\, f)$$

Turing's Fixed Point Combinator

To gain some more intuition for fixed point combinators, let's derive a combinator Θ originally discovered by Turing.

We know that $\Theta\, f$ is a fixed point of f, so we have

$$\Theta\, f = f\,(\Theta\, f).$$

We can write the following recursive equation:

$$\Theta = \lambda f.\, f\,(\Theta\, f)$$

Now use the recursion removal trick:

$$\begin{array}{lcl} \Theta' & \triangleq & \lambda t.\, \lambda f.\, f\,((t\; t)\, f) \\ \Theta & \triangleq & ((\Theta')(\Theta')) \end{array}$$

θ Example

$\mathsf{FACT} = \Theta\, G$

θ Example

$$\begin{aligned} \mathsf{FACT} &= \Theta\, G \\ &= ((\lambda t.\, \lambda f.\, f\,(t\,t\,f))\,(\lambda t.\, \lambda f.\, f\,(t\,t\,f)))\, G \end{aligned}$$

θ Example

$$
\begin{aligned}
\mathsf{FACT} &= \Theta\, G \\
&= ((\lambda t.\, \lambda f.\, f\,(t\,t\,f))\,(\lambda t.\, \lambda f.\, f\,(t\,t\,f)))\, G \\
&\to (\lambda f.\, f\,((\lambda t.\, \lambda f.\, f\,(t\,t\,f))\,(\lambda t.\, \lambda f.\, f\,(t\,t\,f))\, f))\, G
\end{aligned}
$$

θ Example

$$\begin{aligned}
\mathsf{FACT} &= \Theta\, G \\
&= ((\lambda t.\, \lambda f.\, f\,(t\,t\,f))\,(\lambda t.\, \lambda f.\, f\,(t\,t\,f)))\, G \\
&\to (\lambda f.\, f\,((\lambda t.\, \lambda f.\, f\,(t\,t\,f))\,(\lambda t.\, \lambda f.\, f\,(t\,t\,f))\, f))\, G \\
&\to G\,((\lambda t.\, \lambda f.\, f\,(t\,t\,f))\,(\lambda t.\, \lambda f.\, f\,(t\,t\,f))\, G)
\end{aligned}$$

θ Example

$$
\begin{aligned}
\text{FACT} &= \Theta\, G \\
&= ((\lambda t.\, \lambda f.\, f\,(t\,t\,f))\,(\lambda t.\, \lambda f.\, f\,(t\,t\,f)))\; G \\
&\to (\lambda f.\, f\,((\lambda t.\, \lambda f.\, f\,(t\,t\,f))\,(\lambda t.\, \lambda f.\, f\,(t\,t\,f))\, f))\; G \\
&\to G\,((\lambda t.\, \lambda f.\, f\,(t\,t\,f))\,(\lambda t.\, \lambda f.\, f\,(t\,t\,f))\; G) \\
&= G\,(\Theta\, G)
\end{aligned}
$$

θ Example

$$
\begin{aligned}
\mathsf{FACT} &= \Theta\, G \\
&= ((\lambda t.\, \lambda f.\, f\,(t\,t\,f))\,(\lambda t.\, \lambda f.\, f\,(t\,t\,f)))\; G \\
&\to (\lambda f.\, f\,((\lambda t.\, \lambda f.\, f\,(t\,t\,f))\,(\lambda t.\, \lambda f.\, f\,(t\,t\,f))\,f))\; G \\
&\to G\,((\lambda t.\, \lambda f.\, f\,(t\,t\,f))\,(\lambda t.\, \lambda f.\, f\,(t\,t\,f))\,G) \\
&= G\,(\Theta\, G) \\
&= (\lambda f.\, \lambda n.\, \mathbf{if}\ n = 0\ \mathbf{then}\ 1\ \mathbf{else}\ n \times (f\,(n-1)))\,(\Theta\, G) \\
&\to \lambda n.\, \mathbf{if}\ n = 0\ \mathbf{then}\ 1\ \mathbf{else}\ n \times ((\Theta\, G)\,(n-1)) \\
&= \lambda n.\, \mathbf{if}\ n = 0\ \mathbf{then}\ 1\ \mathbf{else}\ n \times (\mathsf{FACT}\,(n-1))
\end{aligned}
$$

Review: Call-by-Value

Here are the syntax and CBV semantics of λ-calculus:

$$
\begin{aligned}
e &::= x \mid \lambda x.\, e \mid e_1\, e_2 \\
v &::= \lambda x.\, e
\end{aligned}
$$

$$
\frac{e_1 \rightarrow e_1'}{e_1\, e_2 \rightarrow e_1'\, e_2} \qquad \frac{e \rightarrow e'}{v\, e \rightarrow v\, e'}
$$

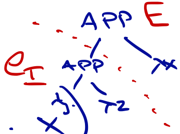
$$
\frac{}{(\lambda x.\, e)\, v \rightarrow e\{v/x\}}\ \beta
$$

There are two kinds of rules: *congruence rules* that specify evaluation order and *computation rules* that specify the "interesting" reductions.

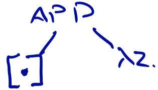
Evaluation Contexts

Evaluation contexts let us separate out these two kinds of rules.

$$e = \big((\lambda y.\ y)\ (\lambda z.\ z)\big)\ (\lambda x.\ x)$$



$$E = [\bullet]\ (\lambda x.\ x)$$



$$E[e_I] = e$$

$$E[e_I] \to E[e_I']$$

$$e_I = (\lambda y.\ y)\ (\lambda z.\ z)$$

$$e_I' = \lambda z. z$$

Evaluation Contexts

Evaluation contexts let us separate out these two kinds of rules.

An evaluation context E is an expression with a "hole" in it: a single occurrence of the special symbol $[\cdot]$ in place of a subexpression.

$$E ::= [\cdot] \mid E\, e \mid v\, E$$

$$((\lambda x.\, x)\ (\lambda x.\, x))\ (\lambda x.\, x)$$

$$E = [\cdot]\ (\lambda x.\, x)$$

$$E \neq ((\lambda x.\, x)\ (\lambda x.\, x))\ [\cdot]$$

Evaluation Contexts

Evaluation contexts let us separate out these two kinds of rules.

An evaluation context E is an expression with a "hole" in it: a single occurrence of the special symbol $[\cdot]$ in place of a subexpression.

$$E ::= [\cdot] \mid E\ e \mid v\ E$$

We write $E[e]$ to mean the evaluation context E where the hole has been replaced with the expression e.

Examples

$$E_1 = [\cdot]\,(\lambda x.\, x)$$
$$E_1[\lambda y.\, y\, y] = (\lambda y.\, y\, y)\; \lambda x.\, x$$

Examples

$$\begin{aligned} E_1 &= [\cdot]\,(\lambda x.\, x) \\ E_1[\lambda y.\, y\, y] &= (\lambda y.\, y\, y)\ \lambda x.\, x \end{aligned}$$

$$\begin{aligned} E_2 &= (\lambda z.\, z\, z)\,[\cdot] \\ E_2[\lambda x.\, \lambda y.\, x] &= (\lambda z.\, z\, z)\,(\lambda x.\, \lambda y.\, x) \end{aligned}$$

Examples

$$
\begin{aligned}
E_1 &= [\cdot]\,(\lambda x.\, x)\\
E_1[\lambda y.\, y\, y] &= (\lambda y.\, y\, y)\ \lambda x.\, x
\end{aligned}
$$

$$
\begin{aligned}
E_2 &= (\lambda z.\, z\, z)\,[\cdot]\\
E_2[\lambda x.\, \lambda y.\, x] &= (\lambda z.\, z\, z)\,(\lambda x.\, \lambda y.\, x)
\end{aligned}
$$

$$
\begin{aligned}
E_3 &= ([\cdot]\ \lambda x.\, x\, x)\,((\lambda y.\, y)\,(\lambda y.\, y))\\
E_3[\lambda f.\, \lambda g.\, f\, g] &= ((\lambda f.\, \lambda g.\, f\, g)\ \lambda x.\, x\, x)\,((\lambda y.\, y)\,(\lambda y.\, y))
\end{aligned}
$$

CBV With Evaluation Contexts

With evaluation contexts, we can define the evaluation semantics for the CBV λ-calculus with just two rules: one for evaluation contexts, and one for β-reduction.

CBV With Evaluation Contexts

With evaluation contexts, we can define the evaluation semantics for the CBV λ-calculus with just two rules: one for evaluation contexts, and one for β-reduction.

With this syntax:

$$E ::= [\cdot] \mid E\, e \mid v\, E$$

$\cancel{(\lambda x.\, \lambda y.\, x)}$

The small-step rules are:

$$\frac{e \to e'}{E[e] \to E[e']}$$

$(\lambda x.\, x)\ (\lambda y.\, y)$

$\to \lambda y.\, y$

$$\frac{}{(\lambda x.\, e)\, v \to e\{v/x\}}\ \beta$$

$E = [\cdot]$

$e =$

CBN With Evaluation Contexts

We can also define the semantics of CBN λ-calculus with evaluation contexts.

CBN With Evaluation Contexts

We can also define the semantics of CBN λ-calculus with evaluation contexts.

For call-by-name, the syntax for evaluation contexts is different:

$$E ::= [\cdot] \mid E\ e$$

CBN With Evaluation Contexts

We can also define the semantics of CBN λ-calculus with evaluation contexts.

For call-by-name, the syntax for evaluation contexts is different:

$$E ::= [\cdot] \mid E\, e$$

But the small-step rules are the same:

$$\frac{e \rightarrow e'}{E[e] \rightarrow E[e']}$$

$$\frac{}{(\lambda x.\, e)\, e' \rightarrow e\{e'/x\}}\ \beta$$

$$\big((\lambda x.\ x)\ (\lambda y.y)\big)\ \big((\lambda z.z)\ (\lambda a.a)\big)$$

$$\rightarrow\ \overbrace{(\lambda y.y)}^{e_0}\ \big((\lambda z.z)\ (\lambda a.a)\big)$$

$E = (\lambda y.y)\ [\circ]$

$e = (\lambda z.z)(\lambda a.a)$

CBV

$$(\lambda y.y)\ (\lambda a.a)$$

$E = [\circ]$

CBN $e =$

$$(\lambda z.z)\ (\lambda a.a)$$

$$\frac{e_0 = \overbrace{e \rightarrow e'}^{\beta} = e_0'}{E[e] \rightarrow E[e']}\ \text{CONTEXT}$$

CBN With Evaluation Contexts

We can also define the semantics of CBN λ-calculus with evaluation contexts.

For call-by-name, the syntax for evaluation contexts is different:

$$E ::= [\cdot] \mid E\; e$$

But the small-step rules are the same:

$$\frac{e \to e'}{E[e] \to E[e']}$$

$$\frac{}{(\lambda x.\, e)\; e' \to e\{e'/x\}}\;\beta$$