CS 4110

Programming Languages & Logics

Lecture 17
Programming in the λ-calculus

10 October 2014

Announcements

- Foster Office Hours 11-12
- Enjoy fall break!

Review: Church Booleans

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

$$\text{TRUE} \triangleq \lambda x.\ \lambda y.\ x$$

$$\text{FALSE} \triangleq \lambda x.\ \lambda y.\ y$$

$$\text{IF} \triangleq \lambda b.\ \lambda t.\ \lambda f.\ b\ t\ f$$

Review: Church Booleans

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

$$\text{TRUE} \triangleq \lambda x.\, \lambda y.\, x$$

$$\text{FALSE} \triangleq \lambda x.\, \lambda y.\, y$$

$$\text{IF} \triangleq \lambda b.\, \lambda t.\, \lambda f.\, b\ t\ f$$

It is easy to see that

$$\text{IF TRUE}\ v\ v' \Downarrow v$$

and

$$\text{IF FALSE}\ v\ v' \Downarrow v'$$

Church Numerals

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

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

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:

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

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:

$$\begin{aligned} \text{SUCC} &\triangleq \lambda n.\, \lambda f.\, \lambda x.\, f\,(n\, f\, x) \\ \text{PLUS} &\triangleq \lambda n_1.\, \lambda n_2.\, n_1\ \text{SUCC}\ n_2 \\ \text{TIMES} &\triangleq \lambda n_1.\, \lambda n_2.\, n_1\ \text{PLUS}\ \mathrm{n}_2\ \text{ZERO} \end{aligned}$$

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:

$$
\begin{aligned}
\text{SUCC} &\triangleq \lambda n.\, \lambda f.\, \lambda x.\, f\,(n\, f\, x) \\
\text{PLUS} &\triangleq \lambda n_1.\, \lambda n_2.\, n_1\ \text{SUCC}\ n_2 \\
\text{TIMES} &\triangleq \lambda n_1.\, \lambda n_2.\, n_1\ \text{PLUS}\ \mathrm{n}_2\ \text{ZERO} \\
\text{ISZERO} &\triangleq \lambda n.\, n\,(\lambda z.\, \textbf{false})\ \textbf{true}
\end{aligned}
$$

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...

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

Recursive Functions

How would we write recursive functions like factorial?

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

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

In slightly more readable notation this is...

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

...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 preveous slide.

Recursion removal trick

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

Define a new function FACT′:

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

Recursion removal trick

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

Define a new function FACT′:

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

Then define FACT as FACT′ applied to itself:

$$\text{FACT} \triangleq \text{FACT}'\ \text{FACT}'$$

Example

Let’s try evaluating FACT on 3...

Example

Let’s try evaluating FACT on 3...

FACT 3

Example

Let's try evaluating FACT on 3...

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

Example

Let's try evaluating FACT on 3...

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

Example

Let's try evaluating FACT on 3...

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

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...

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

Example

Let's try evaluating FACT on 3...

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

Example

Let's try evaluating FACT on 3...

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

Example

Let's try evaluating FACT on 3...

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

Example

Let's try evaluating FACT on 3...

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

So we have a technique for writing recursive functions: write a function f' that takes itself as an argument and define f as $f'\ f'$.

Fixpoint combinators

There is another way of writing recursive functions... we can express the recursive function as the fixed point of some other, higher-order function, and then take its fixed point.

Fixpoint combinators

There is another way of writing recursive functions... we can express the recursive function as the fixed point of some other, higher-order function, and then take its fixed point.

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 we have $G\, g = g$.

Fixpoint combinators

There is another way of writing recursive functions... we can express the recursive function as the fixed point of some other, higher-order function, and then take its fixed point.

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 we have $G\,g = g$.

There are a number of “fixed point combinators,” such as the Y combinator. Thus, we can define the factorial function FACT to be simply Y G, the fixed point of G.

Y Combinator

The (infamous) Y combinator is defined as

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

Y Combinator

The (infamous) Y combinator is defined as

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

It was discovered by Haskell Curry, and is one of the simplest fixed-point combinators.

Y Combinator

The (infamous) Y combinator is defined as

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

It was discovered by Haskell Curry, and is one of the simplest fixed-point combinators.

Note how similar its defnition is to omega:

$$\mathrm{omega} \triangleq (\lambda x.\, x\,x)\,(\lambda x.\, x\,x)$$

Z Combinator

What happens when we evaluate Y *G* under CBV?

Z Combinator

What happens when we evaluate Y *G* under CBV?

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

Z Combinator

What happens when we evaluate Y G under CBV?

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

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

Example

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

Example

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

FACT

Example

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

$$
\begin{aligned}
& \text{FACT} \\
= \quad & \text{Z}\ G
\end{aligned}
$$

Example

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

$$
\begin{aligned}
& \text{FACT} \\
= \quad & \text{Z}\, G \\
= \quad & (\lambda f.\, (\lambda x.\, f\, (\lambda y.\, x\, x\, y))\, (\lambda x.\, f\, (\lambda y.\, x\, x\, y)))\, G
\end{aligned}
$$

Example

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

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

Example

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

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

Example

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

$$
\begin{array}{rl}
 & \text{FACT} \\
= & \mathrm{Z}\, G \\
= & (\lambda f.\, (\lambda x.\, f\, (\lambda y.\, x\, x\, y))\, (\lambda x.\, f\, (\lambda y.\, x\, x\, y)))\, G \\
\rightarrow & (\lambda x.\, G\, (\lambda y.\, x\, x\, y))\, (\lambda x.\, G\, (\lambda y.\, x\, x\, y)) \\
\rightarrow & G\, (\lambda y.\, (\lambda x.\, G\, (\lambda y.\, x\, x\, y))\, (\lambda x.\, G\, (\lambda y.\, x\, x\, y))\, y) \\
= & (\lambda f.\, \lambda n.\, \textbf{if}\ n = 0\ \textbf{then}\ 1\ \textbf{else}\ n \times (f\, (n-1)))
\end{array}
$$

Example

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

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

Example

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

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

Example

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

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

Example

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

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

Example

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

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

Example

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

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

Other fixpoint 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 Fixpoint Combinator

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

Turing's Fixpoint Combinator

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

Suppose we have a higher order function *f*, and want the fixed point of *f*.

Turing's Fixpoint Combinator

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

Suppose we have a higher order function f, and want the fixed point of f.

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

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

Turing's Fixpoint Combinator

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

Suppose we have a higher order function f, and want the fixed point of f.

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

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

This means, that we can write the following recursive equation:

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

Turing's Fixpoint Combinator

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

Suppose we have a higher order function f, and want the fixed point of f.

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

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

This means, that we can write the following recursive equation:

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

Now use the recursion removal trick:

$$\begin{aligned} \Theta' &\triangleq \lambda t.\ \lambda f.\ f\ (t\ t\ f) \\ \Theta &\triangleq \Theta'\ \Theta' \end{aligned}$$

Example

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

Example

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

Example

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

Example

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

Example

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

Example

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

Example

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

Example

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

Definitional Translation

We have seen how to encode a number of high-level language constructs—booleans, conditionals, natural numbers, and recursion—in λ-calculus.

Definitional Translation

We have seen how to encode a number of high-level language constructs—booleans, conditionals, natural numbers, and recursion—in λ-calculus.

In definitional translation, where we define the meaning of language constructs by translation to another language.

Definitional Translation

We have seen how to encode a number of high-level language constructs—booleans, conditionals, natural numbers, and recursion—in λ-calculus.

In definitional translation, where we define the meaning of language constructs by translation to another language.

This is a form of denotational semantics, but instead of the target being mathematical objects, it is a simpler programming language (such as λ-calculus).

Definitional Translation

We have seen how to encode a number of high-level language constructs—booleans, conditionals, natural numbers, and recursion—in λ-calculus.

In definitional translation, where we define the meaning of language constructs by translation to another language.

This is a form of denotational semantics, but instead of the target being mathematical objects, it is a simpler programming language (such as λ-calculus).

For each language construct, we define an operational semantics directly, and then give an alternate semantics by translation to a simpler language.

Review: Call-by-Value

Recall the syntax and CBV semantics of λ-calculus:

$$e ::= x \mid \lambda x.\, e \mid e_1\ e_2$$
$$v ::= \lambda x.\, e$$

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

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

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

Evaluation Contexts

Evaluation contexts are a simple mechanism that separates out these two kinds of rules.

Evaluation Contexts

Evaluation contexts are a simple mechanism that separates out these two kinds of rules.

An evaluation context E (sometimes written $E[\cdot]$) is an expression with a “hole” in it, that is with a single occurrence of the special symbol $[\cdot]$ (called the “hole”) in place of a subexpression.

Evaluation Contexts

Evaluation contexts are a simple mechanism that separates out these two kinds of rules.

An evaluation context E (sometimes written $E[\cdot]$) is an expression with a "hole" in it, that is with a single occurrence of the special symbol $[\cdot]$ (called the "hole") in place of a subexpression.

Evaluation contexts are defined using a BNF grammar that is similar to the grammar used to define the language.

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

Evaluation Contexts

Evaluation contexts are a simple mechanism that separates out these two kinds of rules.

An evaluation context E (sometimes written $E[\cdot]$) is an expression with a "hole" in it, that is with a single occurrence of the special symbol $[\cdot]$ (called the "hole") in place of a subexpression.

Evaluation contexts are defined using a BNF grammar that is similar to the grammar used to define the language.

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

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

Examples

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

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

$$E_2 = (\lambda z.\,z\,z)\,[\cdot]$$
$$E_2[\lambda x.\,\lambda y.\,x] = (\lambda z.\,z\,z)\,(\lambda x.\,\lambda y.\,x)$$

$$E_3 = ([\cdot]\ \lambda x.\,x\,x)\,((\lambda y.\,y)\,(\lambda y.\,y))$$
$$E_3[\lambda f.\,\lambda g.\,f\,g] = ((\lambda f.\,\lambda g.\,f\,g)\ \lambda x.\,x\,x)\,((\lambda y.\,y)\,(\lambda y.\,y))$$

CBV With Evaluation Contexts

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

First we define the contexts:

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

CBV With Evaluation Contexts

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

First we define the contexts:

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

Then we define the small-step rules:

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

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

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.

First we define the contexts:

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

CBN With Evaluation Contexts

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

First we define the contexts:

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

Then we define the small-step rules:

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

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

Multiple Arguments

Our syntax for functions only allows function with a single argument: $\lambda x.\, e$.

Multiple Arguments

Our syntax for functions only allows function with a single argument: $\lambda x.\, e$.

We can define a language that allows functions to have multiple arguments.

$$e ::= x \mid \lambda x_1, \ldots, x_n.\, e \mid e_0\ e_1\ \ldots\ e_n$$

Here, a function $\lambda x_1, \ldots, x_n.\, e$ takes n arguments, with names x_1 through x_n. In a multi argument application $e_0\ e_1\ \ldots\ e_n$, we expect e_0 to evaluate to an n-argument function, and $e_1, \ldots, e_n$ are the arguments that we will give the function.

Multiple Arguments

We can define a CBV operational semantics for the multi-argument λ-calculus as follows.

$$E ::= [\cdot] \mid v_0 \ \ldots \ v_{i-1} \, E \, e_{i+1} \ \ldots \ e_n$$

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

$$\frac{}{(\lambda x_1, \ldots, x_n.\, e_0) \, v_1 \ \ldots \ v_n \rightarrow e_0\{v_1/x_1\}\{v_2/x_2\} \ldots \{v_n/x_n\}} \ \beta$$

Note that the evaluation contexts ensure that we evaluate multi-argument applications $e_0 \; e_1 \ \ldots \ e_n$ from left to right.

Definitional Translation

The multi-argument λ-calculus isn't any more expressive that the pure λ-calculus.

Definitional Translation

The multi-argument λ-calculus isn't any more expressive that the pure λ-calculus.

We can define a translation $\mathcal{T}[\![\cdot]\!]$ that takes an expression in the multi-argument λ-calculus and returns an equivalent expression in the pure λ-calculus.

Definitional Translation

The multi-argument λ-calculus isn't any more expressive that the pure λ-calculus.

We can define a translation $\mathcal{T}[\![\cdot]\!]$ that takes an expression in the multi-argument λ-calculus and returns an equivalent expression in the pure λ-calculus.

$$
\begin{aligned}
\mathcal{T}[\![x]\!] &= x \\
\mathcal{T}[\![\lambda x_1, \ldots, x_n.\, e]\!] &= \lambda x_1.\ \ldots\ \lambda x_n.\, \mathcal{T}[\![e]\!] \\
\mathcal{T}[\![e_0\ e_1\ e_2\ \ldots\ e_n]\!] &= (\ldots((\mathcal{T}[\![e_0]\!]\ \mathcal{T}[\![e_1]\!])\ \mathcal{T}[\![e_2]\!]) \ldots\ \mathcal{T}[\![e_n]\!])
\end{aligned}
$$

Currying

This process of rewriting a function that takes multiple arguments as a chain of functions that each take a single argument is called *currying*.

Currying

This process of rewriting a function that takes multiple arguments as a chain of functions that each take a single argument is called *currying*.

Consider a mathematical function that takes two arguments, the first from domain A and the second from domain B, and returns a result from domain C.

Currying

This process of rewriting a function that takes multiple arguments as a chain of functions that each take a single argument is called *currying*.

Consider a mathematical function that takes two arguments, the first from domain A and the second from domain B, and returns a result from domain C.

We can describe this function, using mathematical notation for function domains, as an element of $A \times B \to C$.

Currying

This process of rewriting a function that takes multiple arguments as a chain of functions that each take a single argument is called *currying*.

Consider a mathematical function that takes two arguments, the first from domain A and the second from domain B, and returns a result from domain C.

We can describe this function, using mathematical notation for function domains, as an element of $A \times B \to C$.

Currying this function produces an element of $A \to (B \to C)$.

Currying

This process of rewriting a function that takes multiple arguments as a chain of functions that each take a single argument is called *currying*.

Consider a mathematical function that takes two arguments, the first from domain A and the second from domain B, and returns a result from domain C.

We can describe this function, using mathematical notation for function domains, as an element of $A \times B \to C$.

Currying this function produces an element of $A \to (B \to C)$.

That is, the curried version takes an argument from domain A, and returns a function that takes an argument from domain B and produces a result of domain C.