

# Encodings

CS 152 (Spring 2024)

Harvard University

Thursday, February 15, 2024

# Today, we will learn about

- ▶ Lambda calculus encodings
- ▶ Church numerals
- ▶ Recursion and fixed point-combinators

# Lambda calculus encodings

- ▶ The pure lambda calculus contains only functions as values.
- ▶ It is not exactly easy to write large or interesting programs in the pure lambda calculus.
- ▶ We can however encode objects, such as booleans, and integers.

# Booleans

# Booleans

We want to define functions *TRUE*, *FALSE*, *AND*, *IF*, and other operators such that the expected behavior holds, for example:

# Booleans

We want to define functions *TRUE*, *FALSE*, *AND*, *IF*, and other operators such that the expected behavior holds, for example:

*AND TRUE FALSE = FALSE*

*IF TRUE e<sub>1</sub> e<sub>2</sub> = e<sub>1</sub>*

*IF FALSE e<sub>1</sub> e<sub>2</sub> = e<sub>2</sub>*

# TRUE and FALSE

# TRUE and FALSE

$TRUE \triangleq \lambda x. \lambda y. x$

$FALSE \triangleq \lambda x. \lambda y. y$



IF

# IF

The function *IF* should behave like

$$\lambda b. \lambda t. \lambda f. \text{if } b = \text{TRUE} \text{ then } t \text{ else } f.$$

The definitions for *TRUE* and *FALSE* make this very easy.

$$IF \triangleq \lambda b. \lambda t. \lambda f. b t f$$

NOT, AND, OR

# NOT, AND, OR

$NOT \triangleq \lambda b. b \text{ FALSE } TRUE$

$AND \triangleq \lambda b_1. \lambda b_2. b_1 \ b_2 \ \text{FALSE}$

$OR \triangleq \lambda b_1. \lambda b_2. b_1 \ \text{TRUE} \ b_2$

# Church numerals

# Church numerals

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

$$\bar{0} \triangleq \lambda f. \lambda x. x$$

$$\bar{1} = \lambda f. \lambda x. f \ x$$

$$\bar{2} = \lambda f. \lambda x. f \ (f \ x)$$

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

# Addition

# Addition

Let us define addition now. Intuitively, the natural number  $n_1 + n_2$  is the result of apply the successor function  $n_1$  times to  $n_2$ .

$$ADD \triangleq \lambda n_1. \lambda n_2. n_1 \text{ SUCC } n_2$$



# Recursion and the fixed-point combinators

# Recursion and the fixed-point combinators

We would like to define a function that computes factorials.

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

# Recursion and the fixed-point combinators

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

# Recursion and the fixed-point combinators

Note that this is not a definition, it's a recursive equation.

# Recursion Removal Trick

- ▶ We can perform a “trick” to define a function *FACT* that satisfies the recursive equation above.
- ▶ First, let's define a new function *FACT'* that looks like *FACT*, but takes an additional argument *f*.
- ▶ We assume that the function *f* will be instantiated with an actual parameter of... *FACT'*.

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

Now we can define the factorial function  $FACT$  in terms of  $FACT'$ .

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

Let's try evaluating  $FACT\ 3 = m$ .

$$\begin{aligned}m &= (FACT'\ FACT')\ 3 \\&= ((\lambda f. \lambda n. \mathbf{if}\ n = 0\ \mathbf{then}\ 1\ \mathbf{else}\ n \times (f\ f\ (n - 1))))\ FACT'\ 3 \\&\longrightarrow (\lambda n. \mathbf{if}\ n = 0\ \mathbf{then}\ 1\ \mathbf{else}\ n \times (FACT'\ FACT'\ (n - 1)))\ 3 \\&\longrightarrow \mathbf{if}\ 3 = 0\ \mathbf{then}\ 1\ \mathbf{else}\ 3 \times (FACT'\ FACT'\ (3 - 1)) \\&\longrightarrow 3 \times (FACT'\ FACT'\ (3 - 1)) \\&\longrightarrow \dots \\&\longrightarrow 3 \times 2 \times 1 \times 1 \\&\longrightarrow^* 6\end{aligned}$$



So we now have a technique for writing a recursive function  $f$ : write a function  $f'$  that explicitly takes a copy of itself as an argument, and then define

$$f \triangleq f' f'.$$

# Fixed point combinators

Alternatively, we can express a recursive function as the fixed point of some other, higher-order function, and then find that fixed point.

# Fixed point combinator

Thus *FACT* is a fixed point of the following function.

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

# Fixed point combinator

Recall that if  $g$  is a fixed point of  $G$ , then we have  $G g = g$ .

# Fixed point combinator

- ▶ A *combinator* is simply a closed lambda term
- ▶ Our functions *SUCC* and *ADD* are examples of combinators.
- ▶ It is possible to define programs using only combinators, thus avoiding the use of variables completely.

# The $Y$ combinator

# The $Y$ combinator

The  $Y$  combinator is defined as

$$Y \triangleq \lambda f. (\lambda x. f (x x)) (\lambda x. f (x x)).$$

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

The fixed point of the higher order function  $G$  is equal to  $G (G (G (G (G \dots))))$ . Intuitively, the  $Y$  combinator unrolls this equality, as needed.



Let's see it in action, on our function  $G$ , where

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

and the factorial function is the fixed point of  $G$ .  
(We will use CBN semantics.)

$$\begin{aligned}
FACT &= Y\ G \\
&= (\lambda f. (\lambda x. f\ (x\ x))\ (\lambda x. f\ (x\ x)))\ G \\
&\longrightarrow (\lambda x. G\ (x\ x))\ (\lambda x. G\ (x\ x)) \\
&\longrightarrow G\ ((\lambda x. G\ (x\ x))\ (\lambda x. G\ (x\ x))) \\
&=_{\beta} G\ (FACT) \\
&= (\lambda f. \lambda n. \mathbf{if}\ n = 0\ \mathbf{then}\ 1\ \mathbf{else}\ n \times (f\ (n - 1)))\ FACT \\
&\longrightarrow \lambda n. \mathbf{if}\ n = 0\ \mathbf{then}\ 1\ \mathbf{else}\ n \times (FACT\ (n - 1))
\end{aligned}$$

Note that the  $Y$  combinator works under CBN semantics, but not CBV. (What happens when we evaluate  $Y G$  under CBV?)

There is a variant of the  $Y$  combinator,  $Z$ , that works under CBV semantics. It is defined as

$$Z \triangleq \lambda f. (\lambda x. f (\lambda y. x x y)) (\lambda x. f (\lambda y. x x y)).$$

# The Turing fixed-point combinator

The Turing fixed-point combinator, denoted  $\Theta$ , was discovered by Alan Turing.

# The Turing fixed-point combinator

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 for  $\Theta$ .

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

Now we can use the recursion removal trick we described earlier! Let's define

$\Theta' = \lambda t. \lambda f. f (t t f)$ , and define

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

Let's try out the Turing combinator on our higher order function  $G$  that we used to define  $FACT$ . Again, we will use CBN semantics.

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



