Programming in the Untyped λ-Calculus

Church & Scott Encodings, Y Combinator

Ernest Ng

CIS 6700, Feb 6th 2023

Motivation

The λ-calculus provides simple semantics for understanding functional abstraction.

Motivation

The λ-calculus provides simple semantics for understanding functional abstraction.

We can encode data purely within the untyped λ-calculus!

Remarks & notational conventions

- Function application is left-associative:

$$\text{Write } t_1\ t_2\ t_3 \text{ to denote } (t_1\ t_2)\ t_3$$

Remarks & **notational conventions**

- Function application is left-associative:

$$\text{Write } t_1\ t_2\ t_3 \text{ to denote } (t_1\ t_2)\ t_3$$

- Bodies of lambda abstractions extend as far right as possible:

$$\text{Write } \lambda x.\ \lambda y.\ x\ y\ x \text{ to denote } \lambda x.\ (\lambda y.\ ((x\ y)\ x))$$

Remarks & notational conventions

- Function application is left-associative:

$$\text{Write } t_1\ t_2\ t_3 \text{ to denote } (t_1\ t_2)\ t_3$$

- Bodies of lambda abstractions extend as far right as possible:

$$\text{Write } \lambda x.\ \lambda y.\ x\ y\ x \text{ to denote } \lambda x.\ (\lambda y.\ ((x\ y)\ x))$$

- A term with no free variables is **closed**
- Closed terms are called **combinators**
  - Simplest combinator: the identity function `id`

$$\mathtt{id} = \lambda x.\ x$$

Agenda

1. Encoding simple datatypes
    - Church Booleans
    - Pairs
2. Church numerals
    - Arithmetic operations
    - Predecessor
    - Testing equality
3. Y-combinator & recursion
    - Factorial
4. Scott encodings
    - Church vs Scott numerals
    - Chruch vs Scott lists

Encoding simple datatypes

Church Booleans

Definition

Let True and False be represented by:

$$tru = \lambda t.\, \lambda f.\, t$$
$$fls = \lambda t.\, \lambda f.\, f$$

Note: tru & fls are normal forms!

Church Booleans

Definition

Let $\mathtt{True}$ and $\mathtt{False}$ be represented by:

$$\mathtt{tru} = \lambda t.\, \lambda f.\, t$$
$$\mathtt{fls} = \lambda t.\, \lambda f.\, f$$

Note: $\mathtt{tru}$ & $\mathtt{fls}$ are normal forms!

Definition

The $\mathtt{test}$ combinator tests the truth value of a Boolean:

$$\mathtt{test} = \lambda l.\, \lambda m.\, \lambda n.\, l\ m\ n$$
$$\mathtt{test}\ \ \mathtt{tru}\, v\, w \to v$$
$$\mathtt{test}\ \ \mathtt{fls}\, v\, w \to w$$

The `test` combinator

Observe:

$$test\ b\ v\ w \longrightarrow\ b\ v\ w$$

The `test` combinator

Observe:

$$\mathtt{test}\; b\; v\; w \longrightarrow\; b\; v\; w$$

Example: (β-redexes underlined)

$$\begin{aligned}
\mathtt{test}\;\mathtt{tru}\; v\; w &\to \underline{(\lambda l.\; \lambda m.\; \lambda n.\; l\; m\; n)\;\mathtt{tru}}\; v\; w \\
&\to \underline{(\lambda m.\; \lambda n.\; \mathtt{tru}\; m\; n)\; v}\; w \\
&\to \underline{(\lambda n.\; \mathtt{tru}\; v\; n)\; w} \\
&\to \mathtt{tru}\; v\; w
\end{aligned}$$

The `test` combinator (cont.)

Observe:

$$\mathtt{test}\ \mathtt{tru}\ v\ w \longrightarrow v$$

$$\text{“if true then v else w”} \longrightarrow v$$

Example: (β-redexes are underlined)

$$\begin{aligned} \mathtt{test}\ \mathtt{tru}\ v\ w &\to \dots \\ &\to \mathtt{tru}\ v\ w \\ &\to \underline{(\lambda t.\, \lambda f.\, t)\ v}\ w \\ &\to \underline{(\lambda f.\, v)\ w} \\ &\to v \end{aligned}$$

Similarly, $\mathtt{test}\ \mathtt{fls}\ v\ w \longrightarrow w$.
(“if false then v else w” $\longrightarrow$ w)

Conjunction

Intuition: $and\ b\ c \approx$ "if b then c else false"

Definition

$$and = \lambda b. \lambda c.\ b\ c\ fls$$

Conjunction

Intuition: $\mathit{and}\; b\; c \approx$ "if b then c else false"

Definition

$$\mathtt{and} = \lambda b.\, \lambda c.\, b\; c\; \mathtt{fls}$$

For Boolean values b, c, we have that:

$$\mathtt{and}\;\; b\;\; c = \begin{cases} c & \text{if } b = \mathtt{tru} \\ b & \text{if } b = \mathtt{fls} \end{cases}$$

Conjunction

Intuition: $and\ b\ c \approx$ "if b then c else false"

Definition

$$and = \lambda b.\, \lambda c.\, b\ c\ fls$$

For Boolean values b, c, we have that:

$$and\ b\ c = \begin{cases} c & \text{if } b = tru \\ b & \text{if } b = fls \end{cases}$$

Examples:

$$\begin{aligned} and\ tru\ b &\to tru\ b\ fls \\ &\to b \end{aligned}$$

$$\begin{aligned} and\ fls\ b &\to fls\ b\ fls \\ &\to fls \end{aligned}$$

Disjunction

Intuition: $or\ b\ c \approx$ "if b then true else c"

Definition

$$or = \lambda b. \lambda c. b\ tru\ c$$

Disjunction

Intuition: $or\ b\ c \approx$ "if b then true else c"

Definition

$$or = \lambda b.\, \lambda c.\, b\ tru\ c$$

Examples:

$$\begin{aligned} or\ tru\ b &\to tru\ tru\ b \\ &\to tru \end{aligned}$$

$$\begin{aligned} or\ fls\ b &\to fls\ tru\ b \\ &\to b \end{aligned}$$

Negation

Intuition: $\mathit{not}\ b \approx$ "if b then false else true"

Definition

$$\mathit{not} = \lambda b.\, b\ \mathit{fls}\ \mathit{tru}$$

Negation

Intuition: $not\ b \approx$ "if b then false else true"

Definition

$$not = \lambda b.\ b\ fls\ tru$$

$$\begin{aligned} not\ tru &\to (\lambda b.\ b\ fls\ tru)\ tru \\ &\to tru\ fls\ tru \\ &\to fls \end{aligned}$$

$$\begin{aligned} not\ fls &\to (\lambda b.\ b\ fls\ tru)\ fls \\ &\to fls\ fls\ tru \\ &\to tru \end{aligned}$$

Pairs

Intuition: $(v, w) \approx$ "$\lambda b.$ if b then v else w"

$$\begin{aligned} \mathtt{pair} &= \lambda v.\, \lambda w.\, \lambda b.\, b\ v\ w \\ \implies \mathtt{pair}\ \ v\ \ w &= \lambda b.\, b\ v\ w \end{aligned}$$

Pairs

Intuition: $(v, w) \approx$ "$\lambda b.$ if b then v else w"

$$
\begin{aligned}
\mathit{pair} &= \lambda v.\, \lambda w.\, \lambda b.\, b\; v\; w \\
\implies \mathit{pair}\;\; v\;\; w &= \lambda b.\, b\; v\; w
\end{aligned}
$$

When applied to a Boolean b, $\mathit{pair}\;\; v\;\; w$ applies b to v and w:

$$
\begin{aligned}
\mathit{pair}\;\; v\;\; w\;\; \mathit{tru} &\to \mathit{tru}\;\; v\;\; w \\
&\to v
\end{aligned}
$$

$$
\begin{aligned}
\mathit{pair}\;\; v\;\; w\;\; \mathit{fls} &\to \mathit{fls}\;\; v\;\; w \\
&\to w
\end{aligned}
$$

Pairs

Intuition: $(v, w) \approx$ "$\lambda b.$ if b then v else w"

$$\begin{aligned} \mathtt{pair} &= \lambda v.\, \lambda w.\, \lambda b.\, b\ v\ w \\ \Longrightarrow \mathtt{pair}\ v\ w &= \lambda b.\, b\ v\ w \end{aligned}$$

When applied to a Boolean b, $\mathtt{pair}\ v\ w$ applies b to v and w:

$$\begin{aligned} \mathtt{pair}\ v\ w\ \mathtt{tru} &\to \mathtt{tru}\ v\ w \\ &\to v \end{aligned}$$

$$\begin{aligned} \mathtt{pair}\ v\ w\ \mathtt{fls} &\to \mathtt{fls}\ v\ w \\ &\to w \end{aligned}$$

This motivates the projection functions $\mathtt{fst}$ & $\mathtt{snd}$:

$$\begin{aligned} \mathtt{fst} &= \lambda p.\, p\ \mathtt{tru} \\ \mathtt{snd} &= \lambda p.\, p\ \mathtt{fls} \end{aligned}$$

Pairs (cont.)

Example: (β-redexes underlined)

$$\begin{aligned}
\mathtt{fst}\ (\mathtt{pair}\ \mathtt{v}\ \mathtt{w}) &\to \mathtt{fst}\ (\lambda b.\, b\, v\, w) \\
&\to \underline{(\lambda p.\, p\ \mathtt{tru})\, (\lambda b.\, b\, v\, w)} \quad \text{(by definition of } \mathtt{fst}\text{)} \\
&\to \underline{(\lambda b.\, b\, v\, w)\ \mathtt{tru}} \\
&\to \mathtt{tru}\ \mathtt{v}\ \mathtt{w} \\
&\to v
\end{aligned}$$

Church numerals

Church numerals

Intuition: "A number *n* is a function that does something *n* times"

Church numerals

Intuition: "A number n is a function that does something n times"

Definition

Define the **Church numerals** $c_0, c_1, c_2, \ldots$ as follows:

$$\begin{aligned} c_0 &= \lambda s.\, \lambda z.\, z \\ c_1 &= \lambda s.\, \lambda z.\, s\ z \\ c_2 &= \lambda s.\, \lambda z.\, s\ (s\ z) \\ &\ldots \end{aligned}$$

Church numerals

Intuition: "A number n is a function that does something n times"

Definition

Define the **Church numerals** $c_0, c_1, c_2, \ldots$ as follows:

$$
\begin{aligned}
c_0 &= \lambda s.\, \lambda z.\, z \\
c_1 &= \lambda s.\, \lambda z.\, s\ z \\
c_2 &= \lambda s.\, \lambda z.\, s\ (s\ z) \\
&\ \ldots
\end{aligned}
$$

Each $n \in \mathbb{N}$ is represented by a combinator c_n that takes arguments s and z ("successor" and "zero") and applies s to z for n times.

$$c_n = \lambda s.\, \lambda z.\ \langle \text{apply } s \text{ to } z \text{ for } n \text{ times} \rangle$$

Successor function

Definition

The **successor function** scc on Church numerals is defined as:

$$scc = \lambda n.\ \lambda s.\ \lambda z.\ s\ (n\ s\ z)$$

Successor function

Definition

The **successor function** scc on Church numerals is defined as:

$$scc = \lambda n.\ \lambda s.\ \lambda z.\ s\ (n\ s\ z)$$

Intuition: $n + 1 \approx$ "apply s to z for n times, then apply s once more"

scc takes a Church numeral n and returns $\underbrace{\text{another Church numeral}}_{\substack{\text{function that takes } s, z \\ \text{\& applies } s \text{ repeatedly to } z}}$

Successor function (cont.)

Example: showing that “$scc\ 0 = 1$”:

$$
\begin{aligned}
scc\ \ c_0 &\to \underbrace{(\lambda n.\, \lambda s.\, \lambda z.\, s\ (n\ s\ z))}_{scc}\ \ \underbrace{(\lambda s.\, \lambda z.\, z)}_{c_0} \\
&\to\ \lambda s.\, \lambda z.\, s\ (\underbrace{(\lambda s.\, \lambda z.\, z)}_{c_0}\ \ s\ z) \\
&\to\ \lambda s.\, \lambda z.\, s\ (\underbrace{(\lambda z.\, z)}_{id}\ \ z) \\
&\to\ \lambda s.\, \lambda z.\, s\ z \\
&= c_1 && \text{(by definition of } c_1\text{)}
\end{aligned}
$$

Successor function (cont.)

Another way* to define the successor function:

$$scc_2 = \lambda n.\ \lambda s.\ \lambda z.\ n\ s\ (s\ z)$$

Intuition: "apply s to $(s\ \ z)$ for n times"

(as opposed to "applying s to z for $(n + 1)$ times")

*TAPL Exercise 5.2.2

Addition of Church numerals

$$\begin{aligned} \mathtt{plus} &= \lambda m.\, \lambda n.\, \lambda s.\, \lambda z.\; m\; s\; (n\; s\; z) \\ \Longrightarrow \underbrace{\mathtt{plus}\; m\; n}_{m+n} &= \lambda s.\, \lambda z.\; m\; s\; (n\; s\; z) \end{aligned}$$

Addition of Church numerals

$$\begin{aligned} \mathtt{plus} &= \lambda m.\, \lambda n.\, \lambda s.\, \lambda z.\, m\ s\ (n\ s\ z) \\ \Longrightarrow \underbrace{\mathtt{plus}\ m\ n}_{m+n} &= \lambda s.\, \lambda z.\, m\ s\ (n\ s\ z) \end{aligned}$$

Intuition: To compute $m + n$,

1. $\underbrace{\text{Apply } s \text{ iterated } n \text{ times to } z}_{n\ s\ z}$ ...
2. ... then $\underbrace{\text{apply } s \text{ to the result for } m \text{ more times}}_{m\ s\ (n\ s\ z)}$

Addition (cont.)

Recall: $c_1 = \lambda s.\, \lambda z.\, s\ z$

Example: Proving 1 + 1 = 2

$$\begin{aligned} \mathtt{plus}\ c_1\ c_1 &\to \lambda s.\, \lambda z.\, c_1\ s\ \underline{(c_1\ s\ z)} \\ &\to \lambda s.\, \lambda z.\, \underline{c_1\ s\ (s\ z)} \\ &\to \lambda s.\, \lambda z.\, s\ (s\ z) \\ &= c_2 \qquad \text{(by definition of } c_2\text{)} \end{aligned}$$

Multiplication of Church numerals

Definition

$$\mathtt{times} = \lambda m.\, \lambda n.\, m\, (\mathtt{plus}\ \ n)\, c_0$$

$$m\, (\mathtt{plus}\ \ n)\, c_0 \approx \text{“apply } \mathtt{plus}\ n \text{ iterated } m \text{ times to } c_0 \text{ (zero)”}$$

$$\approx \text{“add together } m \text{ copies of } n\text{”}$$

Multiplication (cont.)

Can we define multiplication without using *plus*? Recall that:

$$\mathtt{times}\ m\ n \approx \text{“add together } m \text{ copies of } n\text{”}$$

*TAPL Exercise 5.2.3

*Here, $n\ s$ is akin to $\mathtt{plus}\ n$

Multiplication (cont.)

Can we define multiplication without using $\mathtt{plus}$? Recall that:

$$\mathtt{times}\ m\ n \approx \text{“add together } m \text{ copies of } n\text{”}$$

This motivates an alternate definition*:

$$\mathtt{times} = \lambda m.\ \lambda n.\ \lambda s.\ \lambda z.\ m\ (n\ s)\ z$$

Intuition: $m\ (n\ s)\ z \approx$ “apply $(n\ s)$ to z for m times”*

*TAPL Exercise 5.2.3

*Here, $n\ s$ is akin to $\mathtt{plus}\ n$

Multiplication example

$$\mathtt{times} = \lambda x.\, \lambda y.\, \lambda a.\, x\ (y\ a)$$

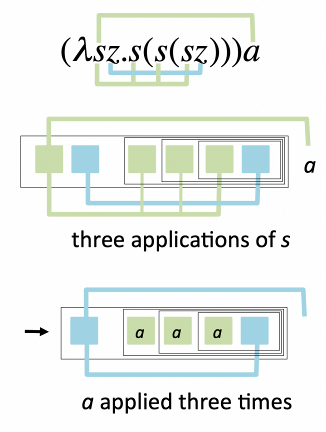
Compute 3×3:

$$\begin{aligned} \mathtt{times}\ c_3\ c_3 &= (\lambda x.\, \lambda y.\, \lambda a.\, x\ (y\ a))\ c_3\ c_3 \\ &\to (\lambda a.\, c_3\ (c_3\ a)) \end{aligned}$$

Multiplication example (cont.)

Consider the term (c_3 a):

$$c_3 = \lambda s.\, \lambda z.\, s\ (s\ (s\ z))$$



Applying c_3 to a produces a function that applies a three times (Rojas)

Multiplication example (cont.)

Let **3a** denote $(c_3\ a)$. Now, consider c_3 (**3a**):

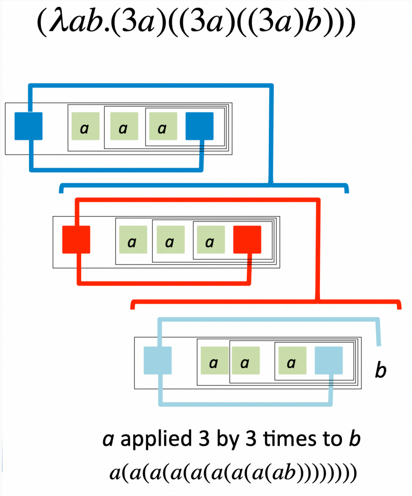
$$\lambda a.\ c_3\ (\mathbf{3a}) = \left(\lambda a.\ \underbrace{(\lambda s.\ \lambda b.\ s\ (s\ (s\ b)))}_{c_3}\ (\mathbf{3a})\right)$$

$$\rightarrow \lambda a.\ \lambda b.\ \mathbf{3a}\ (\mathbf{3a}\ (\mathbf{3a}\ b))$$

Applying c_3 to **3a** returns a function that applies **3a** three times
= applies a for (3 × 3) times

Example from Rojas (2015), *A Tutorial Introduction to the Lambda Calculus*

Multiplication example (cont.)



c_3 applied to **3a**, visualized

Diagram from Rojas (2015), *A Tutorial Introduction to the Lambda Calculus*

How should we define `predecessor` for Church numerals?

Predecessor function

Strategy: Create a pair $(n - 1, n)$, then pick the 1st element of the pair

Predecessor function

Strategy: Create a pair $(n-1, n)$, then pick the 1st element of the pair

We define two auxiliary functions:

$$zz = \mathtt{pair}\ c_0\ c_0$$
$$ss = \lambda p.\ \mathtt{pair}\ (\mathtt{snd}\ p)\ (\mathtt{plus}\ c_1\ (\mathtt{snd}\ p))$$

When applied to a pair (i, j), $\mathtt{ss}$ returns a pair $(j, j+1)$:

$$\mathtt{ss}\ \left(\mathtt{pair}\, c_i\ c_j\right) = \mathtt{pair}\ c_j\ c_{j+1}$$

Predecessor function

Strategy: Create a pair $(n-1, n)$, then pick the 1st element of the pair

We define two auxiliary functions:

$$zz = \mathtt{pair}\ c_0\ c_0$$
$$ss = \lambda p.\ \mathtt{pair}\ (\mathtt{snd}\ p)\,(\mathtt{plus}\ c_1\ (\mathtt{snd}\ p))$$

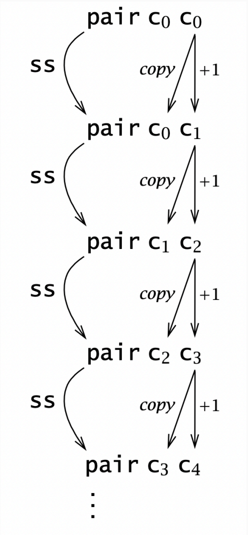
When applied to a pair (i, j), $\mathtt{ss}$ returns a pair $(j, j+1)$:

$$\mathtt{ss}\ \left(\mathtt{pair}\, c_i\ c_j\right) = \mathtt{pair}\ c_j\ c_{j+1}$$

The predecessor function $\mathtt{prd}$ involves applying $\mathtt{ss}$ to $\mathtt{pair}\, c_0 c_0$ for m times, then projecting the 1st component:

$$\mathtt{prd} = \lambda m.\ \mathtt{fst}\,(m\ \mathtt{ss}\ \mathtt{zz})$$

Predecessor function



(diagram from TAPL)

$prd \approx$ "apply ss to $pair\ c_0\ c_0$ for m times"

$$\approx \begin{cases} pair\ \ c_0\ c_0 & \text{when } m = 0 \\ pair\ \ c_{m-1}\ c_m & \text{otherwise} \end{cases}$$

Evaluating $prd\ c_n$ requires $O(n)$ steps!

5-minute break

Roadmap for the next few slides

Aim: To represent *factorial* in the untyped λ-calculus

To do this, we need to discuss the following:

1. Testing if a Church numeral $\stackrel{?}{=}$ 0
2. Equality of Church numerals
3. Y-comabintor & recursion

Testing if a Church numeral $\stackrel{?}{=}$ 0

Definition

$$\mathtt{isZero} = \lambda m.\, m\, (\lambda x.\, \mathtt{fls})\, \mathtt{tru}$$

Example (β-redexes underlined):

$$\begin{aligned}
\mathtt{isZero}\, c_0 &= (\lambda m.\, m\, (\lambda x.\, \mathtt{fls})\, \mathtt{tru})\, c_0 \\
&= \underline{(\lambda m.\, m\, (\lambda x.\, \mathtt{fls})\, \mathtt{tru})\, (\lambda s.\, \lambda z.\, z)} \quad \text{(by definition of } c_0\text{)} \\
&\to \underline{(\lambda s.\, \lambda z.\, z)\, (\lambda x.\, \mathtt{fls})}\, \mathtt{tru} \\
&\to \underline{(\lambda z.\, z)\, \mathtt{tru}} \\
&\to \mathtt{tru}
\end{aligned}$$

Equality of Church numerals

Intuition: $m == n \iff (m - n) == 0 \wedge (n - m) == 0$

Definition

The *equal* function tests two Church numerals for equality, returning a Church Boolean:

```
equal = λm. λn.
          and (isZero (m prd n))
            (isZero (n prd m))
```

m prd n ≈ "applying the predecessor function for *m* times on *n*"

≈ "m *minus* n"

Y-combinator & recursion

How do we represent recursion?

Ω-combinator

Definition

The **divergent combinator** Ω is:

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

Definition

The **divergent combinator** Ω is:

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

Let's try to β-reduce Ω:

$$\begin{aligned}(\lambda x.\ x\ x)\ (\lambda x.\ x\ x) &\to (x\ x)\big[\,x \coloneqq (\lambda x.\ x\ x)\,\big] \\ &\to (\lambda x.\ x\ x)\ (\lambda x.\ x\ x)\end{aligned}$$

We get what we started with!

A λ-term is **divergent** if it has no β-normal form.

Y-combinator

Definition

The **fixpoint combinator** is the term

$$\mathbf{Y} = \lambda f.\, (\lambda x.\, f\ (x\ x))\ (\lambda x.\, f\ (x\ x))$$

Definition

The **fixpoint combinator** is the term

$$\mathbf{Y} = \lambda f.\, (\lambda x.\, f\ (x\ x))\, (\lambda x.\, f\ (x\ x))$$

$$\begin{aligned}\mathbf{Y}\ F &= \Big(\lambda f.\, (\lambda x.\, f\ (x\ x))\, (\lambda x.\, f\ (x\ x))\Big)\ F \\ &\to (\lambda x.\, F\ (x\ x))\, (\lambda x.\, F\ (x\ x)) \\ &\to F\ \Big(\underbrace{(\lambda x.\, F\ (x\ x))\, (\lambda x.\, F\ (x\ x))}_{\mathbf{Y}\ F}\Big) \\ &\to F\ (\mathbf{Y}\ F)\end{aligned}$$

Y-combinator

Definition

The **fixpoint combinator** is the term

$$\mathbf{Y} = \lambda f.\, (\lambda x.\, f\ (x\ x))\ (\lambda x.\, f\ (x\ x))$$

$$\begin{aligned}
\mathbf{Y}\ F &= \Big(\lambda f.\, (\lambda x.\, f\ (x\ x))\ (\lambda x.\, f\ (x\ x))\Big)\ F \\
&\to (\lambda x.\, F\ (x\ x))\ (\lambda x.\, F\ (x\ x)) \\
&\to F\ \Big(\underbrace{(\lambda x.\, F\ (x\ x))\ (\lambda x.\, F\ (x\ x))}_{\mathbf{Y}\ F}\Big) \\
&\to F\ (\mathbf{Y}\ F)
\end{aligned}$$

Say that $\mathbf{Y}\ F$ is a **fixed point** of the function F:

$$\mathbf{Y}\ F = F\ (\mathbf{Y}\ F)$$

Y-combinator

We can use **Y** to achieve recursive calls to *F*:

$$\begin{aligned} \mathbf{Y}\ F &= F\ (\mathbf{Y}\ F) \\ &= F\ (F\ (\mathbf{Y}\ F)) \\ &= \ldots \end{aligned}$$

Factorial

Definition

Using Church numerals, we define the factorial function as:

$$
\begin{aligned}
fact = \lambda f.\,\lambda n.\; & \mathit{if}\ \ \mathit{isZero}\ n\ \ \mathit{then}\ c_1 \\
& \mathit{else}\ \ \mathit{times}\ n\ \big(f\,(\mathit{prd}\ n)\big)
\end{aligned}
$$

where $n \in \mathbb{N}$ & f is the function to call in the body

Factorial (cont.)

Use **Y** to achieve recursive calls to *fact*:

$$(\mathbf{Y}\ fact)\ c_1 = (fact\ (\mathbf{Y}\ fact))\ c_1$$

$$\rightarrow if\ equal\ c_1\ c_0\ then\ c_1\ else\ times\ c_1\ \left((\mathbf{Y}\ fact)\ c_0\right)$$

$$\rightarrow times\ c_1\ \left((\mathbf{Y}\ fact)\ c_0\right)$$

$$\rightarrow times\ c_1\ \left(fact\ (\mathbf{Y}\ fact)\ c_0\right)$$

$$\rightarrow times\ c_1\ \left(if\ equal\ c_0\ c_0\ then\ c_1\right.$$

$$\left.else\ times\ c_0\ ((\mathbf{Y}\ fact)\ (prd\ c_0))\right)$$

$$\rightarrow times\ c_1\ c_1$$

$$\rightarrow c_1$$

Factorial (cont.)

Instead of using the Y-combinator, we can also define factorial using the U-combinator. ⏩ (see appendix)

Scott encodings

Consider the following algebraic data types in Haskell:

$$\mathit{data\ Nat = Zero \mid Succ\ Nat}$$
$$\mathit{data\ List\ a = Nil \mid Cons\ a\ (List\ a)}$$

Scott encodings

Consider the following algebraic data types in Haskell:

```
data Nat = Zero | Succ Nat
data List a = Nil | Cons a (List a)
```

Scott encodings allow us to encode ADTs as λ-terms.

Scott numerals

Definition

$$\begin{aligned} \mathit{zero} &= \lambda z.\, \lambda s.\, z \\ \mathit{scc} &= \lambda n.\, \lambda z.\, \lambda s.\, s\; n \end{aligned}$$

Intuition: Arguments distinguish between different cases

Church vs Scott numerals

How do the Church & Scott encodings differ?

Church vs Scott numerals

How do the Church & Scott encodings differ?

Church	Scott
$\mathit{zero} = \lambda s.\ \lambda z.\ z$	$\mathit{zero} = \lambda z.\ \lambda s.\ z$
$\mathit{scc} = \lambda n.\ \lambda s.\ \lambda z.\ s\ (n\ s\ z)$	$\mathit{scc} = \lambda n.\ \lambda z.\ \lambda s.\ s\ n$

Church vs Scott numerals

Church	Scott
$scc = \lambda n.\, \lambda s.\, \lambda z.\, s\ (n\ s\ z)$	$scc = \lambda n.\, \lambda z.\, \lambda s.\, s\ n$
folds continuation threaded throughout structure	*case analysis* continuation unwraps one layer only

Church vs Scott numerals

Church	Scott
$\lambda s.\, \lambda z.\, z$ $\lambda s.\, \lambda z.\, s\ z$ $\lambda s.\, \lambda z.\, s\ (s\ z)$ $\lambda s.\, \lambda z.\, s\ (s\ (s\ z))$	$\lambda z.\, \lambda s.\, z$ $\lambda z.\, \lambda s.\, s\ (\lambda s.\, \lambda z.\, z)$ $\lambda z.\, \lambda s.\, s\ (\lambda s.\, \lambda z.\, s\ (\lambda s.\, \lambda z.\, z))$ $\lambda z.\, \lambda s.\, s\ (\lambda s.\, \lambda z.\, s\ (\lambda s.\, \lambda z.\, s\ (\lambda s.\, \lambda z. z)))$
"apply s, iterated n times"	"apply s on the preceding Scott numeral"

Church vs Scott encodings: Predecessor

Church: $O(n)$	**Scott**: $O(1)$
$prd = \lambda m.\ \mathtt{fst}\,(m\ ss\ zz)$ where $zz = \mathtt{pair}\ c_0\ c_0$, $ss = \lambda p.\ \mathtt{pair}\ (\mathtt{snd}\ p)\ (\mathtt{plus}\ c_1\,(\mathtt{snd}\ p))$	$prd = \lambda n.\ n\ \mathtt{zero}\,(\lambda p.\ p)$

Predecessor can be expressed more succintly using Scott encodings!

Church encoding for lists

Definition

$$\mathit{nil} = \lambda n.\, \lambda c.\, n$$

$$\mathit{cons} = \lambda x.\, \lambda l.\, \lambda n.\, \lambda\, c.\, c\; x\; (l\; n\; c)$$

(akin to *foldr*)

Church encoding for lists

Definition

$$\mathtt{nil} = \lambda n.\, \lambda c.\, n$$

$$\mathtt{cons} = \lambda x.\, \lambda l.\, \lambda n.\, \lambda\, c.\, c\; x\; (l\; n\; c)$$

(akin to $\mathtt{foldr}$)

$x \approx$ “head”

$l \approx$ “tail”

$n \approx$ case for $\mathtt{nil}$

$c \approx$ case for $\mathtt{cons}$

Church encoding for lists (cont.)

Definition

$$\mathtt{nil} = \lambda n.\, \lambda c.\, n$$

$$\mathtt{cons} = \lambda x.\, \lambda l.\, \lambda n.\, \lambda\, c.\, c\ x\ (l\ n\ c)$$

(akin to *foldr*)

Example:

$$x : y : z : [\,] \quad \approx \quad \lambda c.\, \lambda n.\, (c\ x\ (c\ y\ (c\ z\ n)))$$

Scott encoding for lists

Definition

$$\mathtt{nil} = \lambda n.\, \lambda c.\, n$$

$$\mathtt{cons} = \lambda x.\, \lambda l.\, \lambda n.\, \lambda c.\, c\ x\ l$$

$x \approx$ "head"

$\mathtt{l} \approx$ "tail"

$n \approx$ case for $\mathtt{nil}$

$c \approx$ case for $\mathtt{cons}$

Church vs Scott lists

Church	Scott
$cons = \lambda x.\, \lambda l.\, \lambda n.\, \lambda\, c.\; c\; x\; (l\; n\; c)$	$cons = \lambda x.\, \lambda l.\, \lambda n.\, \lambda c.\; c\; x\; l$ (much simpler!)

$x \approx$ "head"

$l \approx$ "tail"

$n \approx$ case for nil

$c \approx$ case for $cons$

Church vs Scott encodings

- Encodings only differ for recursive datatypes

Church vs Scott encodings

- Encodings only differ for recursive datatypes
- **Church**: defines how functions should be folded over an element of the type

Church vs Scott encodings

- Encodings only differ for recursive datatypes
- **Church**: defines how functions should be folded over an element of the type
- **Scott**: uses “case analysis”, recursion not immediately visible

Church vs Scott encodings

- Encodings only differ for recursive datatypes
- **Church**: defines how functions should be folded over an element of the type
- **Scott**: uses “case analysis”, recursion not immediately visible
  - Simpler representation (for certain functions)
  - **Y**-combinator needed for other operations

Further reading:
Jansen (2013), *Programming in the λ-Calculus: From Church to Scott and Back*

References i

- Foster, Jeff (Nov. 2017). *Lambda Calculus Encodings.* *https://www.cs.umd.edu/class/fall2017/cmsc330/lectures/02-lambda-calc-encodings.pdf*.
- Geuvers, Herman (2014). *The Church-Scott representation of inductive and coinductive data.* *http://www.cs.ru.nl/~herman/PUBS/ChurchScottDataTypes.pdf*.
- Jansen, Jan Martin (Jan. 2013). “Programming in the λ-Calculus: From Church to Scott and Back”. In: DOI: *10.1007/978-3-642-40355-2_12*.
- Pierce, Benjamin C. (2002). *Types and Programming Languages.* 1st. The MIT Press. ISBN: 0262162091.
- Rojas, Raúl (2015). “A Tutorial Introduction to the Lambda Calculus”. In: *CoRR* abs/1503.09060. arXiv: *1503.09060*. URL: *http://arxiv.org/abs/1503.09060*.

References ii

Sampson, Adrian (Jan. 2018). *λ-Calculus Encodings*. *https://www.cs.cornell.edu/courses/cs6110/2019sp/lectures/lec03.pdf*.

Selinger, Peter (2008). "Lecture notes on the lambda calculus". In: *CoRR* abs/0804.3434. arXiv: *0804.3434*. URL: *http://arxiv.org/abs/0804.3434*.

Appendix

Appendix: Defining `factorial` using the U-combinator

Instead of using the **Y**-combinator, we can also define *`factorial`* using the **U**-combinator.

Definition
The **U**-combinator applies its argument f to itself:

$$\mathbf{U} = \lambda f.\ f\ f$$

Appendix: Defining `factorial` using the U-combinator

Recall the definition of factorial:

$$fact = \lambda f. \lambda n. \mathit{if}\ \mathit{equal}\ n\ c_0\ \mathit{then}\ c_1$$
$$\mathit{else}\ \mathit{times}\ n\ \left(f\ (\mathit{prd}\ n)\right)$$

See this link for worked examples

Appendix: Defining `factorial` using the U-combinator

Recall the definition of factorial:

$$\begin{aligned} fact = \lambda f.\lambda n.\, & if\ equal\ n\ c_0\ then\ c_1 \\ & else\ times\ n\ \left(f\,(prd\ n)\right) \end{aligned}$$

We can define factorial using **U** as follows:

$$\begin{aligned} fact = \mathbf{U}\Big(\lambda f.\lambda n.\, & if\ isZero\ n\ then\ c_1 \\ & else\ times\ n\ \left(\mathbf{U}\,f\,(prd\ n)\right)\Big) \end{aligned}$$

See this link for worked examples

Appendix: More on the U-combinator

It turns out that we can define **Y** using **U**:

$$\mathbf{U} = \lambda f.\, f\ f$$

$$\mathbf{Y} = \lambda g.\, \mathbf{U}\left(\lambda f.\, g\ (\underline{\mathbf{U}\ f})\right)$$

$$\rightarrow \lambda g.\, \underline{\mathbf{U}\left(\lambda f.\, g\ (f\ f)\right)}$$

$$\rightarrow \underbrace{\lambda g.\, \left(\lambda f.\, g\ (f\ f)\right)\left(\lambda f.\, g\ (f\ f)\right)}_{\text{definition of } \mathbf{Y} \text{ we saw on slide 32 (up to } \alpha\text{-equivalence)}}$$

definition of **Y** we saw on slide 32
(up to α-equivalence)

◂ Back to main presentation

Appendix: CBV vs CBN

- **Call-by-value** (CBV): only reduce outermost redexex, and given an application $(\lambda x.\ e_1)\ e_2$, make sure e_2 is a *value* before applying the abstraction
    - Reduce a redex only when its RHS has already been reduced to a value
- **Call-by-name** (CBN): Reduce the leftmost, outermost redex first, but we *don't* allow reductions inside abstractions
- TAPL & these slides both use CBV.

◂ Back to main presentation