

Programming in the Untyped λ -Calculus

Church & Scott Encodings, Y Combinator

Ernest Ng

CIS 6700, Feb 6th 2023

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

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:

Write $t_1 t_2 t_3$ to denote $(t_1 t_2) t_3$

Remarks & notational conventions

- Function application is left-associative:

Write $t_1 t_2 t_3$ to denote $(t_1 t_2) t_3$

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

Write $\lambda x. \lambda y. x y x$ to denote $\lambda x. (\lambda y. ((x y) x))$

Remarks & notational conventions

- Function application is left-associative:

Write $t_1 t_2 t_3$ to denote $(t_1 t_2) t_3$

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

Write $\lambda x. \lambda y. x y x$ 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

$$id = \lambda x. x$$

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

- Church 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 *True* and *False* be represented by:

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

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

Note: *tru* & *fls* are normal forms!

Definition

The *test* combinator tests the truth value of a Boolean:

$$test = \lambda l. \lambda m. \lambda n. l\ m\ n$$

$$test\ tru\ v\ w \rightarrow v$$

$$test\ fls\ v\ w \rightarrow w$$

The test combinator

Observe:

$$\text{test } b \, v \, w \longrightarrow b \, v \, w$$

The test combinator

Observe:

$$\text{test } b \, v \, w \longrightarrow b \, v \, w$$

Example: (β -redexes underlined)

$$\begin{aligned} \text{test } tru \, v \, w &\rightarrow \underline{(\lambda l. \lambda m. \lambda n. l \, m \, n) \, tru \, v \, w} \\ &\rightarrow \underline{(\lambda m. \lambda n. tru \, m \, n) \, v \, w} \\ &\rightarrow \underline{(\lambda n. tru \, v \, n) \, w} \\ &\rightarrow tru \, v \, w \end{aligned}$$

The test combinator (cont.)

Observe:

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

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

Example: (β -redexes are underlined)

$$\text{test } \text{tru} \ v \ w \rightarrow \dots$$

$$\rightarrow \text{tru} \ v \ w$$

$$\rightarrow \underline{(\lambda t. \lambda f. t) \ v \ w}$$

$$\rightarrow \underline{(\lambda f. v) \ w}$$

$$\rightarrow v$$

Similarly, $\text{test } \text{fls} \ v \ w \longrightarrow w$.

("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: $and\ b\ c \approx$ “if b then c else false”

Definition

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

For Boolean values b, c , we have that:

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

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:

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

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

Disjunction

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

Definition

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

Disjunction

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

Definition

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

Examples:

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

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

Negation

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

Definition

$$\text{not} = \lambda b. b \text{ fls } \text{tru}$$

Negation

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

Definition

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

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

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

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

$$pair = \lambda v. \lambda w. \lambda b. b \ v \ w$$

$$\implies pair \ v \ w = \lambda b. b \ v \ w$$

Pairs

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

$$\text{pair} = \lambda v. \lambda w. \lambda b. b \ v \ w$$

$$\implies \text{pair } v \ w = \lambda b. b \ v \ w$$

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

$$\begin{aligned} \text{pair } v \ w \ \text{tru} &\rightarrow \text{tru } v \ w \\ &\rightarrow v \end{aligned}$$

$$\begin{aligned} \text{pair } v \ w \ \text{fls} &\rightarrow \text{fls } v \ w \\ &\rightarrow w \end{aligned}$$

Pairs

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

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

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

$$\begin{aligned} \text{pair } v \ w \ \text{tru} &\rightarrow \text{tru } v \ w \\ &\rightarrow v \end{aligned}$$

$$\begin{aligned} \text{pair } v \ w \ \text{fls} &\rightarrow \text{fls } v \ w \\ &\rightarrow w \end{aligned}$$

This motivates the projection functions fst & snd :

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

Example: (β -redexes underlined)

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

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, \dots as follows:

$$c_0 = \lambda s. \lambda z. z$$

$$c_1 = \lambda s. \lambda z. s\ z$$

$$c_2 = \lambda s. \lambda z. s\ (s\ z)$$

...

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, \dots as follows:

$$c_0 = \lambda s. \lambda z. z$$

$$c_1 = \lambda s. \lambda z. s\ z$$

$$c_2 = \lambda s. \lambda z. s\ (s\ z)$$

...

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.

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

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 another Church numeral
function that takes s, z
& applies s repeatedly to z

Successor function (cont.)

Example: showing that “ $scc\ 0 = 1$ ”:

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

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

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

Addition of Church numerals

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

Intuition: To compute $m + n$,

1. $\underbrace{\text{Apply } s \text{ iterated } n \text{ times to } z \dots}_{n \ s \ z}$
2. \dots 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 $1 + 1 = 2$

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

Multiplication of Church numerals

Definition

$$times = \lambda m. \lambda n. m \ (plus \ n) \ c_0$$

$m \ (plus \ n) \ c_0 \approx$ “apply *plus n* iterated *m* times to c_0 (zero)”
 \approx “add together *m* copies of *n*”

Multiplication (cont.)

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

times m n \approx “add together *m* copies of *n*”

*TAPL Exercise 5.2.3

*Here, *n s* is akin to *plus n*

Multiplication (cont.)

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

$times\ m\ n \approx$ “add together m copies of n ”

This motivates an alternate definition*:

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

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 $plus\ n$

Multiplication example

$$times = \lambda x. \lambda y. \lambda a. x (y a)$$

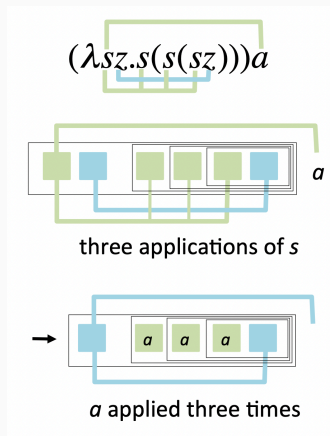
Compute 3×3 :

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

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)

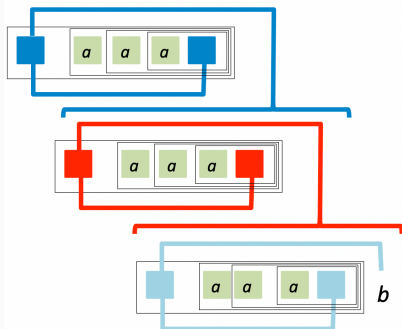
Multiplication example (cont.)

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

$$\begin{aligned}\lambda a. c_3\ (\mathbf{3a}) &= \left(\lambda a. \underbrace{(\lambda s. \lambda b. s\ (s\ (s\ b)))}_{c_3} (\mathbf{3a}) \right) \\ &\rightarrow \lambda a. \lambda b. \mathbf{3a}\ (\mathbf{3a}\ (\mathbf{3a}\ b))\end{aligned}$$

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

Multiplication example (cont.)

$$(\lambda ab.(3a)((3a)((3a)b)))$$


a applied 3 by 3 times to b
 $a(a(a(a(a(a(ab))))))))$

c_3 applied to $3a$, visualized

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:

$$zz = \text{pair } c_0 \ c_0$$

$$ss = \lambda p. \text{pair } (\text{snd } p) (\text{plus } c_1 (\text{snd } p))$$

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

$$ss (\text{pair } c_i \ c_j) = \text{pair } c_j \ c_{j+1}$$

Predecessor function

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

We define two auxiliary functions:

$$zz = \text{pair } c_0 c_0$$

$$ss = \lambda p. \text{pair } (\text{snd } p) (\text{plus } c_1 (\text{snd } p))$$

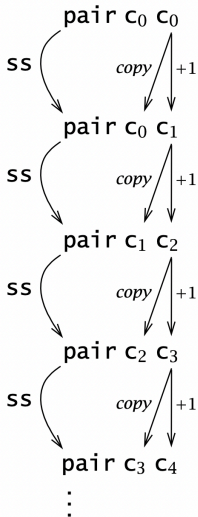
When applied to a pair (i, j) , ss returns a pair $(j, j + 1)$:

$$ss (\text{pair } c_i c_j) = \text{pair } c_j c_{j+1}$$

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

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

Predecessor function



$prd \approx$ “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!

(diagram from TAPL)

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

$$isZero = \lambda m. m (\lambda x. fls) tru$$

Example (β -redexes underlined):

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

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))
```

$m \text{ prd } n \approx$ “applying the predecessor function for m times on n ”

\approx “ m minus n ”

Y-combinator & recursion

How do we represent recursion?

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 :

$$\begin{aligned} (\lambda x. x x) (\lambda x. x x) &\rightarrow (x x) \left[x := (\lambda x. x x) \right] \\ &\rightarrow (\lambda x. x x) (\lambda x. x x) \end{aligned}$$

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

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

Y-combinator

Definition

The **fixpoint combinator** is the term

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

$$\begin{aligned} Y F &= \left(\lambda f. (\lambda x. f (x x)) (\lambda x. f (x x)) \right) F \\ &\rightarrow (\lambda x. F (x x)) (\lambda x. F (x x)) \\ &\rightarrow F \left(\underbrace{(\lambda x. F (x x)) (\lambda x. F (x x))}_{Y F} \right) \\ &\rightarrow F (Y F) \end{aligned}$$

Y-combinator

Definition

The **fixpoint combinator** is the term

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

$$\begin{aligned}\mathbf{Y} F &= \left(\lambda f. (\lambda x. f (x x)) (\lambda x. f (x x)) \right) F \\&\rightarrow (\lambda x. F (x x)) (\lambda x. F (x x)) \\&\rightarrow F \left(\underbrace{(\lambda x. F (x x)) (\lambda x. F (x x))}_{\mathbf{Y} F} \right) \\&\rightarrow F (\mathbf{Y} F)\end{aligned}$$

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

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

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

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

Definition

Using Church numerals, we define the factorial function as:

$$\begin{aligned} fact = \lambda f. \lambda n. & \text{ if equal } n \ c_0 \text{ then } 1 \\ & \text{ else times } n \left(f (prd \ n) \right) \end{aligned}$$

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

Factorial (cont.)

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

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

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

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

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

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

$$\rightarrow \text{times } c_1 c_1$$

$$\rightarrow c_1$$

Scott encodings

Consider the following algebraic data types in Haskell:

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

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.

Definition

$$zero = \lambda z. \lambda s. z$$
$$succ = \lambda n. \lambda z. \lambda s. s \ n$$

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

Church vs Scott numerals

Church	Scott
$scc = \lambda n. \lambda s. \lambda z. s \text{ (} n \text{ } s \text{ } z \text{)}$	$scc = \lambda n. \lambda z. \lambda s. s \text{ } n$
<i>folds</i> continuation threaded throughout structure	<i>case analysis</i> continuation unwraps one layer only

Church vs Scott numerals

Church	Scott
$\lambda s. \lambda z. z$	$\lambda z. \lambda s. z$
$\lambda s. \lambda z. s \text{ } z$	$\lambda z. \lambda s. s (\lambda s. \lambda z. z)$
$\lambda s. \lambda z. s (s \text{ } z)$	$\lambda z. \lambda s. s (\lambda s. \lambda z. s (\lambda s. \lambda z. z))$
$\lambda s. \lambda z. s (s (s \text{ } z))$	$\lambda z. \lambda s. s (\lambda s. \lambda z. s (\lambda s. \lambda z. s (\lambda s. \lambda z. z)))$
“apply s , iterated n times”	“apply s on the preceding Scott numeral”

Church vs Scott encodings: Predecessor

Church: $O(n)$

$prd = \lambda m. fst (m\ ss\ zz)$
where $zz = pair\ c_0\ c_0$
 $ss = \lambda p. pair\ (snd\ p)$
 $\quad (plus\ c_1\ (snd\ p))$

Scott: $O(1)$

$prd = \lambda n. n\ zero\ (\lambda p. p)$

Predecessor can be expressed more succinctly using Scott encodings!

Definition

$$nil = \lambda n. \lambda c. n$$
$$cons = \lambda x. \lambda l. \lambda n. \lambda c. c \ x \ (l \ n \ c)$$

(akin to *foldr*)

Definition

$$nil = \lambda n. \lambda c. n$$
$$cons = \lambda x. \lambda l. \lambda n. \lambda c. c \ x \ (l \ n \ c)$$

(akin to *foldr*)

$x \approx$ “head”

$l \approx$ “tail”

$n \approx$ case for *nil*

$c \approx$ case for *cons*

Church encoding for lists (cont.)

Definition

$$nil = \lambda n. \lambda c. n$$

$$cons = \lambda x. \lambda l. \lambda n. \lambda c. c\ x\ (l\ n\ c)$$

(akin to *foldr*)

Example:

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

Definition

$$nil = \lambda n. \lambda c. n$$
$$cons = \lambda x. \lambda l. \lambda n. \lambda c. c \ x \ l$$
$$x \approx \text{"head"}$$
$$l \approx \text{"tail"}$$
$$n \approx \text{case for } nil$$
$$c \approx \text{case for } cons$$

Church vs Scott lists

Church

$cons = \lambda x. \lambda l. \lambda n. \lambda c. c\ x\ (l\ n\ c)$

Scott

$cons = \lambda x. \lambda l. \lambda n. \lambda c. c\ x\ l$
(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*

-  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”. In.
-  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](https://doi.org/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>.



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>.