Programming Language Theory

Lambda Calculus
Looking back, looking forward

Done: Syntax, semantics, and equivalence

▶ For a language with nothing but loops and global variables

Didn’t IMP leave some things out?

▶ In particular: scope, functions, and data structures
▶ (And also: strings, I/O, exceptions, threads, . . .)

Today: Time for a new model… (Pierce, chapter 5)
Higher-order functions work well for scope and data structures.

- **Scope**: Not all memory/variables available to all code.
  
  **Example**:

  ```haskell
  val x = 1
  fun add y =
    let val z = 2
    in  x + y + z
    end
  val seven = add 3
  ```

- **Data**: Function closures store data.
  
  **Example**: Association “list”

  ```haskell
  val empty = (fn k => raise Empty)
  fun cons k v lst =
    (fn k' => if k'=k then v else lst k')
  fun lookup k lst = lst k
  ```

  (Later: Objects do both too)
Adding data structures

Extending **IMP** with data structures isn’t too hard:

\[
\begin{align*}
    e & ::= c \mid x \mid e + e \mid e * e \mid (e, e) \mid e.1 \mid e.2 \\
    v & ::= c \mid (v, v) \\
    H & ::= \cdot \mid H, x \mapsto v
\end{align*}
\]

\[H; e \Downarrow v\]

\[
\begin{array}{c}
    H; e_1 \Downarrow v_1 \quad H; e_2 \Downarrow v_2 \\
    \hline
    H; (e_1, e_2) \Downarrow (v_1, v_2) \\
    H; e \Downarrow (v_1, v_2) \quad H; e \Downarrow (v_1, v_2)
\end{array}
\]

\[H; e.1 \Downarrow v_1 \quad H; e.2 \Downarrow v_2\]

Note: We allow pairs of values, not just pairs of integers
Adding data structures

Extending \textbf{IMP} with data structures isn’t too hard:

\[
e ::= c | x | e + e | e * e | (e, e) | e.1 | e.2
\]
\[
v ::= c | (v, v)
\]
\[
H ::= · | H, x \mapsto v
\]

Note: We allow pairs of values, not just pairs of integers
Note: We now have \textit{stuck} programs (e.g., \texttt{c.1})


Note: Division also causes stuckness
What about functions

But adding functions (or objects) does not work well:

\[
\begin{align*}
e & ::= \ldots \mid \text{fn } x \Rightarrow s \\
v & ::= \ldots \mid \text{fn } x \Rightarrow s \\
s & ::= \ldots \mid e(e)
\end{align*}
\]

\[
\begin{align*}
H;e \downarrow v \\
H;fn \ x \Rightarrow s \downarrow fn \ x \Rightarrow s \\
H;e_1 \downarrow \text{fn } x \Rightarrow s & \quad H;e_2 \downarrow v \\
H;e_1(e_2) \rightarrow H;x := v ; s
\end{align*}
\]

Does this match "the semantics we want" for function calls?
What about functions

But adding functions (or objects) does not work well:

\[ e ::= \ldots | \text{fn } x \Rightarrow s \]
\[ v ::= \ldots | \text{fn } x \Rightarrow s \]
\[ s ::= \ldots | e(e) \]

Does this match “the semantics we want” for function calls?
What about functions

But adding functions (or objects) does not work well:

\[
\begin{align*}
  e & ::= \ldots \mid \text{fn } x \implies s \\
  v & ::= \ldots \mid \text{fn } x \implies s \\
  s & ::= \ldots \mid e(e)
\end{align*}
\]

NO: Consider \( x := 1 ; (\text{fn } x \implies y := x)(2) ; \text{ans} := x \)
What about functions

But adding functions (or objects) does not work well:

\[
\begin{align*}
e & ::= \ldots \mid \text{fn } x => s \\
v & ::= \ldots \mid \text{fn } x => s \\
s & ::= \ldots \mid e(e)
\end{align*}
\]

\[
\begin{align*}
H;e \Downarrow v \\
H;fn \ x => s \Downarrow fn \ x => s \\
H;e_1 \Downarrow \text{fn } x => s & \quad H;e_2 \Downarrow v \\
H;e_1(e_2) & \rightarrow H;x := v \ ; s
\end{align*}
\]

NO: Consider \( x := 1 \ ; (\text{fn } x => y := x)(2) \ ; \text{ans} := x \)

Scope matters, variable names do not matter. That is:

- Local variables should “be local”
- Choice of local-variable names should have only local ramifications
Another try

\[ H; s \rightarrow H'; s' \]

\[
\frac{H; e_1 \downarrow \text{fn } x \Rightarrow s \quad H; e_2 \downarrow v \quad y \text{ “fresh”}}{H; e_1(e_2) \rightarrow H; y := x \ ; \ x := v \ ; \ s \ ; \ x := y}
\]
Another try

\[ H; s \rightarrow H'; s' \]

\[
\begin{array}{c}
H; e_1 \downarrow \text{fn } x = s & H; e_2 \downarrow v \quad y \text{ “fresh”} \\
\hline
H; e_1(e_2) \rightarrow H; y := x ; x := v ; s ; x := y
\end{array}
\]

- “fresh” isn’t very IMP-like, but okay (think malloc)
- not a good match to how functions are implemented
- NO: wrong model for most functional and OO languages
  - (even wrong for C if s calls another function that accesses the global variable x)
Another try

\[ H; s \rightarrow H'; s' \]

\[ H; e_1 \downarrow \text{fn } x \Rightarrow s \quad H; e_2 \downarrow v \quad y \text{ “fresh”} \]

\[ H; e_1(e_2) \rightarrow H; y := x ; x := v ; s ; x := y \]

\[ f_1 := (\text{fn } x \Rightarrow f_2 := (\text{fn } z \Rightarrow \text{ans} := x + z)) ; \]
\[ f_1(2) ; x := 3 ; f_2(4) \]

“Should” set \text{ans} to 6:

▶ \( f_1(2) \) should assign to \( f_2 \) a function
  that adds 2 to its argument and stores result in \text{ans}.

“Actually” sets \text{ans} to 7:

▶ \( f_1(2) \) assigns to \( f_2 \) a function
  that adds \textit{the current value of} \( x \) to its argument.
Punch line

Cannot properly model local scope via a global heap of integers.

▶ Functions are not syntactic sugar for assignments to globals.

So let’s build a new model that focuses on this essential concept.

▶ (can add other IMP features back later)

Or just borrow a model from Alonzo Church.

And drop mutation, conditionals, integers (!), and loops (!)
The Lambda Calculus

The Lambda Calculus:

\[ e ::= \lambda x. e \mid x \mid e e \]

\[ v ::= \lambda x. e \]

You apply a function by \textit{substituting} the argument for the \textit{bound variable}.

▶ (There is an equivalent \textit{environment} definition not unlike heap-copying; see future homework.)
Example substitutions

\[ e ::= \lambda x. e \mid x \mid e \ e \]
\[ v ::= \lambda x. e \]

Substitution is the key operation we were missing:

\[ (\lambda x. x)(\lambda y. y) \rightarrow (\lambda y. y) \]
\[ (\lambda x. \lambda y. y \ x)(\lambda z. z) \rightarrow (\lambda y. y \ \lambda z. z) \]
\[ (\lambda x. x \ x)(\lambda x. x \ x) \rightarrow (\lambda x. x \ x)(\lambda x. x \ x) \]

After substitution, the bound variable is gone, so its “name” was irrelevant.

▶ (Good!)

There are \textit{irreducible} expressions, like \( x \ z \)

▶ (Bad?)
A programming language

Given substitution \((e_1[e_2/x] = e_3)\), we can give a semantics:

\[
e \rightarrow_{cbv} e'
\]

\[
\frac{e[v/x] = e'}{(\lambda x. e) v \rightarrow_{cbv} e'}
\]

\[
\frac{e_1 \rightarrow_{cbv} e_1'}{e_1 e_2 \rightarrow_{cbv} e_1' e_2}
\]

\[
\frac{e_2 \rightarrow_{cbv} e_2'}{v e_2 \rightarrow_{cbv} v e_2'}
\]

A small-step, call-by-value (CBV), left-to-right operational semantics

- Terminates when the “whole program” is some \(\lambda x. e\)

But (also) gets stuck when there’s a free variable “at top-level”

- (Won’t “cheat” like we did with \(H @ x \leadsto c\) in IMP, because we’re interested in modeling scope)

This is the “heart” of functional languages like SML

- (but “real” implementations don’t substitute; they do something equivalent)
Roadmap

- Motivation for a new model (done)
- CBV lambda calculus using substitution (done (almost))
- Notes on concrete syntax
- Simple Lambda encodings — LC is Turing complete!
- Other reduction strategies
- Defining substitution
Concrete syntax notes

We (and SML) resolve concrete-syntax ambiguities as follows:

1. \( \lambda x. e_1 \ e_2 \) is \( (\lambda x. e_1 \ e_2) \), not \( (\lambda x. e_1) \ e_2 \)

2. \( e_1 \ e_2 \ e_3 \) is \( (e_1 \ e_2) \ e_3 \), not \( e_1 \ (e_2 \ e_3) \)

   (Convince yourself application is not associative)
Concrete syntax notes

We (and SML) resolve concrete-syntax ambiguities as follows:

1. $\lambda x.\ e_1\ e_2$ is $(\lambda x.\ e_1\ e_2)$, not $(\lambda x.\ e_1)\ e_2$

2. $e_1\ e_2\ e_3$ is $(e_1\ e_2)\ e_3$, not $e_1\ (e_2\ e_3)$
   ▶ (Convince yourself application is not associative)

More generally:

1. Function bodies extend to an unmatched right parenthesis
   Example: $(\lambda x.\ y\ (\lambda z.\ z)\ w)\ q$

2. Application associates to the left
   Example: $e_1\ e_2\ e_3\ e_4$ is $(((e_1\ e_2)\ e_3)\ e_4)$
Concrete syntax notes

We (and SML) resolve concrete-syntax ambiguities as follows:

1. \( \lambda x. e_1 e_2 \) is \((\lambda x. e_1) e_2\), not \((\lambda x. e_1 e_2)\)

2. \( e_1 e_2 e_3 \) is \((e_1 e_2) e_3\), not \(e_1 (e_2 e_3)\)
   
   (Convince yourself application is not associative)

More generally:

1. Function bodies extend to an unmatched right parenthesis
   
   Example: \((\lambda x. y (\lambda z. z) w) q\)

2. Application associates to the left
   
   Example: \(e_1 e_2 e_3 e_4\) is \(((e_1 e_2) e_3) e_4\)

   Like in IMP, dealing with abstract syntax trees
   (with non-leaves labeled \(\lambda\) or “application”)

   Rules may seem strange at first, but are the most convenient
   (based on 70 years experience)
The Lambda Calculus

Abstract syntax:

$$e ::= \lambda x. e \mid x \mid e \, e$$
$$\nu ::= \lambda x. e$$

A small-step, call-by-value (CBV), left-to-right operational semantics:

$$e \rightarrow_{cbv} e'$$

$$e[v/x] = e'$$

$$\frac{e \rightarrow_{cbv} e'}{(\lambda x. e) \, v \rightarrow_{cbv} e'}$$

$$\frac{e_1 \rightarrow_{cbv} e_1'}{e_1 \, e_2 \rightarrow_{cbv} e_1' \, e_2}$$

$$\frac{e_2 \rightarrow_{cbv} e_2'}{v \, e_2 \rightarrow_{cbv} v \, e_2'}$$
Lambda encodings

Lambda Calculus appears to be too simple to be useful; left out:

- constants and arithmetic primitives
- conditionals
- data structures
- loops and recursion

In fact, LC is Turing complete and can encode everything else (like assembly language can encode high-level features).

Motivation for encodings:

- Fun and mind-expanding
- Shows we aren’t oversimplifying the model
- (“numbers are just syntactic sugar”)
- Shows some languages are too expressive (e.g., unlimited C++ template instantiation)

Encodings are also just “(re)definition via translation”.

Matthew Fluet
Programming Language Theory
Lecture 07
Lambda encodings

Lambda Calculus appears to be too simple to be useful; left out:
➤ constants and arithmetic primitives
➤ conditionals
➤ data structures
➤ loops and recursion

In fact, LC is *Turing complete* and can *encode* everything else (like assembly language can encode high-level features).
Lambda encodings

Lambda Calculus appears to be too simple to be useful; left out:
- constants and arithmetic primitives
- conditionals
- data structures
- loops and recursion

In fact, LC is Turing complete and can encode everything else (like assembly language can encode high-level features).

Motivation for encodings:
- Fun and mind-expanding
- Shows we aren’t oversimplifying the model
  - (“numbers are just syntactic sugar”)
- Shows some languages are too expressive
  - (e.g., unlimited C++ template instantiation)

Encodings are also just “(re)definition via translation”.
Encoding booleans

The “boolean ADT”:

▶ There are two booleans and one conditional expression.
▶ The conditional takes 3 arguments (via currying).
▶ If the first arg is one boolean, then it evaluates to the second arg.
▶ If the first arg is the other boolean, then it evaluates to the third arg.
▶ Any 3 expressions meeting this specification is an encoding of booleans.

true = λ x. λ y. x
false = λ x. λ y. y
if = λ b. λ t. λ f. b t f

(This is just one encoding.)
Encoding booleans

The “boolean ADT”:

- There are two booleans and one conditional expression.
- The conditional takes 3 arguments (via currying).
  - If the first arg is one boolean, then it evaluates to the second arg.
  - If the first arg is the other boolean, then it evaluates to the third arg.

true = λx. λy. x
false = λx. λy. y
if = λb. λt. λf. b t f

(This is just one encoding.)
Encoding booleans

The “boolean ADT”:

- There are two booleans and one conditional expression.
- The conditional takes 3 arguments (via currying).
  - If the first arg is one boolean, then it evaluates to the second arg.
  - If the first arg is the other boolean, then it evaluates to the third arg.

Any 3 expressions meeting this specification is an encoding of booleans.
Encoding booleans

The “boolean ADT”:

- There are two booleans and one conditional expression.
- The conditional takes 3 arguments (via currying).
  - If the first arg is one boolean, then it evaluates to the second arg.
  - If the first arg is the other boolean, then it evaluates to the third arg.

Any 3 expressions meeting this specification is an encoding of booleans.

\[
\begin{align*}
\text{true} &= \lambda x. \lambda y. x \\
\text{false} &= \lambda x. \lambda y. y \\
\text{if} &= \lambda b. \lambda t. \lambda f. b t f
\end{align*}
\]

(This is just one encoding.)
Encoding booleans

true  =  \lambda x. \lambda y. x
false =  \lambda x. \lambda y. y
if    =  \lambda b. \lambda t. \lambda f. b t f

Example:

if true \nu_1 \nu_2
\equiv (((if true) \nu_1) \nu_2
\equiv (((\lambda b. \lambda t. \lambda f. b t f) (\lambda x. \lambda y. x)) \nu_1) \nu_2
\rightarrow_{cbv} (((\lambda t. \lambda f. (\lambda x. \lambda y. x) t f) \nu_1) \nu_2
\rightarrow_{cbv} (\lambda f. (\lambda x. \lambda y. x) \nu_1 f) \nu_2
\rightarrow_{cbv} (\lambda x. \lambda y. x) \nu_1 \nu_2
\rightarrow_{cbv} (\lambda y. \nu_1) \nu_2
\rightarrow_{cbv} \nu_1
Evaluation order matters

Careful: With CBV we need to “thunk” …

\[
\text{if true } (\lambda x. x) \underbrace{((\lambda x. x x)(\lambda x. x x))}_{\text{an infinite loop}}
\]

diverges, but

\[
\text{if true } (\lambda x. x) \underbrace{(\lambda z. ((\lambda x. x x)(\lambda x. x x)))}_{\text{a value that diverges when called}}
\]

doesn’t.
Evaluation order matters

Careful: With CBV we need to “thunk” ... 

\[
\text{if true } (\lambda x. x) ((\lambda x. x x)(\lambda x. x x)) \] 

an infinite loop

\[ \rightarrow_{\text{cbv}} (\lambda t. \lambda f. \text{true } t f) (\lambda x. x) ((\lambda x. x x)(\lambda x. x x)) \]

\[ \rightarrow_{\text{cbv}} (\lambda f. \text{true } (\lambda x. x) f) ((\lambda x. x x)(\lambda x. x x)) \]

\[ \rightarrow_{\text{cbv}} (\lambda f. \text{true } (\lambda x. x) f) ((\lambda x. x x)(\lambda x. x x)) \]

\[ \rightarrow_{\text{cbv}} \ldots \]

diverges, but

\[
\text{if true } (\lambda x. x) (\lambda z. ((\lambda x. x x)(\lambda x. x x))) \]

a value that diverges when called

\[ \rightarrow_{\text{cbv}} (\lambda t. \lambda f. \text{true } t f) (\lambda x. x) (\lambda z. ((\lambda x. x x)(\lambda x. x x))) \]

\[ \rightarrow_{\text{cbv}} (\lambda f. \text{true } (\lambda x. x) f) (\lambda z. ((\lambda x. x x)(\lambda x. x x))) \]

\[ \rightarrow_{\text{cbv}} \text{true } (\lambda x. x) (\lambda z. ((\lambda x. x x)(\lambda x. x x))) \]

\[ \rightarrow_{\text{cbv}} (\lambda y. (\lambda x. x)) (\lambda z. ((\lambda x. x x)(\lambda x. x x))) \]

\[ \rightarrow_{\text{cbv}} \lambda x. x \]

doesn’t.
Encoding pairs

The “pair ADT”:
▶ There is one constructor (taking two arguments) and two selectors.
▶ The first selector returns the first arg passed to the constructor.
▶ The second selector returns the second arg passed to the constructor.
Encoding pairs

The “pair ADT”:

- There is one constructor (taking two arguments) and two selectors.
- The first selector returns the first arg passed to the constructor.
- The second selector returns the second arg passed to the constructor.

\[
\text{mkpair} = \lambda x. \lambda y. \lambda z. z \, x \, y \\
\text{fst} = \lambda p. p \, (\lambda x. \lambda y. x) \\
\text{snd} = \lambda p. p \, (\lambda x. \lambda y. y)
\]

Example: \( \text{snd} \, (\text{fst} \, (\text{mkpair} \, (\text{mkpair} \, v_1 \, v_2) \, v_3)) \to^{*}_{\text{cbv}} v_2 \)
Reusing lambdas

Is it weird that the encodings of Booleans and pairs both used \( \lambda x. \lambda y. x \) and \( \lambda x. \lambda y. y \) for different purposes?
Reusing lambdas

Is it weird that the encodings of Booleans and pairs both used $\lambda x. \lambda y. x$ and $\lambda x. \lambda y. y$ for different purposes?

Is it weird that the same bit-pattern in binary code can represent an integer, a floating-point, an instruction, or a pointer?

Von Neumann: Bits can represent (all) code and data

Church (?): Lambdas can represent (all) code and data
Encoding lists

Rather than start from scratch, notice that booleans and pairs are enough:

- Empty list: \( \text{nil} = \text{mkpair false false} \)
- Non-empty list: \( \text{cons} = \lambda h. \lambda t. \text{mkpair true (mkpair } h \ t) \)
- Is-empty predicate: \( \ldots \)
- Head: \( \ldots \)
- Tail: \( \ldots \)

(Not too far from how lists are implemented.)
Encoding lists

Rather than start from scratch, notice that booleans and pairs are enough:

- Empty list: \texttt{nil} = \texttt{mkpair false false}
- Non-empty list: \texttt{cons} = \lambda h. \lambda t. \texttt{mkpair true (mkpair h t)}
- Is-empty predicate: \texttt{null} = \lambda l. \texttt{not (fst l)}
- Head: \ldots
- Tail: \ldots

(Not too far from how lists are implemented.)
Encoding lists

Rather than start from scratch, notice that booleans and pairs are enough:

- **Empty list**: \( \text{nil} = \text{mkpair \ false \ false} \)
- **Non-empty list**: \( \text{cons} = \lambda h. \lambda t. \text{mkpair \ true \ (mkpair \ h \ t)} \)
- **Is-empty predicate**: \( \text{null} = \lambda l. \text{not \ (fst \ l)} \)
- **Head**: \( \text{hd} = \lambda l. \text{fst \ (snd \ l)} \)
- **Tail**: \( \ldots \)

(Not too far from how lists are implemented.)
Encoding lists

Rather than start from scratch, notice that booleans and pairs are enough:

- Empty list: \texttt{nil} = \texttt{mkpair false false}
- Non-empty list: \texttt{cons} = \lambda h. \lambda t. \texttt{mkpair true (mkpair h t)}
- Is-empty predicate: \texttt{null} = \lambda l. \texttt{not (fst l)}
- Head: \texttt{hd} = \lambda l. \texttt{fst (snd l)}
- Tail: \texttt{tl} = \lambda l. \texttt{snd (snd l)}

(Not too far from how lists are implemented.)
Encoding lists

Rather than start from scratch, notice that booleans and pairs are enough:

- Empty list: nil = mkpair false false
- Non-empty list: cons = λh. λt. mkpair true (mkpair h t)
- Is-empty predicate: null = λl. not (fst l)
- Head: hd = λl. fst (snd l)
- Tail: tl = λl. snd (snd l)

(Not too far from how lists are implemented.)

What happens with tl (tl nil)?
Encoding lists

Rather than start from scratch, notice that booleans and pairs are enough:

- Empty list: \( \text{nil} = \text{mkpair} \ false \ false \)
- Non-empty list: \( \text{cons} = \lambda h. \lambda t. \text{mkpair} \ true \ (\text{mkpair} \ h \ t) \)
- Is-empty predicate: \( \text{null} = \lambda l. \text{not} \ (\text{fst} \ l) \)
- Head: \( \text{hd} = \lambda l. \text{fst} \ (\text{snd} \ l) \)
- Tail: \( \text{tl} = \lambda l. \text{snd} \ (\text{snd} \ l) \)

(Not too far from how lists are implemented.)

What happens with \( \text{tl} \ (\text{tl} \ \text{nil}) \)?
- Will produce some lambda.
- Just like \( \text{NULL} \to \text{tail} \to \text{tail} \) would produce some bit-pattern.
Encoding recursion

Some programs diverge, but can we write *useful* loops? Yes!

▶ Write a function that takes an \( f \) and calls it in place of recursion

▶ Example (in enriched language):

\[
\lambda f. \lambda x. \text{if } (x = 0) \text{ then } 1 \text{ else } (x \ast f(x - 1))
\]

▶ Then apply \( \text{fix} \) to it to get a recursive function:

\[
\text{fix}(\lambda f. \lambda x. \text{if } (x = 0) \text{ then } 1 \text{ else } (x \ast f(x - 1)))
\]

\( \text{fix} \) reduces to something (roughly) equivalent to

\[
e[(\text{fix}(\lambda f. e)) / f]
\]

which is "unrolling the recursion once" (and further unrollings will happen as necessary).

The details are intricate; the point is that you define \( \text{fix} \) only once.

▶ \( \text{fix}^{\text{cbv}} = \lambda f. (\lambda x. f((\lambda y. x x y)))((\lambda x. f((\lambda y. x x y)))) \)

▶ \( \text{fix}^{\text{cbn}} = \lambda f. (\lambda x. f(x x))((\lambda x. f(x x))) \)

(technical jargon: the \( Y \) combinator(s))
Encoding recursion

Some programs diverge, but can we write *useful* loops? Yes!

To write a recursive function:

- Write a function that takes an $f$ and calls it in place of recursion
  
  Example (in enriched language):
  $$\lambda f. \lambda x. \text{if } (x = 0) \text{ then } 1 \text{ else } (x \ast f(x - 1))$$

- Then apply $\text{fix}$ to it to get a recursive function:
  $$\text{fix} \left( \lambda f. \lambda x. \text{if } (x = 0) \text{ then } 1 \text{ else } (x \ast f(x - 1)) \right)$$

  $\text{fix}$ reduces to something (roughly) equivalent to
  $$e \left( \text{fix} \left( \lambda f. e \right) / f \right),$$
  which is “unrolling the recursion once” (and further unrollings will happen as necessary).

  The details are intricate; the point is that you define $\text{fix}$ only once.

- $\text{fix}_{cbv} = \lambda f. \left( \lambda x. f \left( \lambda y. x x y \right) \right) \left( \lambda x. f \left( \lambda y. x x y \right) \right)$

- $\text{fix}_{cbn} = \lambda f. \left( \lambda x. f \left( x x \right) \right) \left( \lambda x. f \left( x x \right) \right)$

  (technical jargon: the $Y$ combinator(s))
Encoding recursion

Some programs diverge, but can we write *useful* loops? Yes!

To write a recursive function:

- Write a function that takes an $f$ and calls it in place of recursion

  - Example (in enriched language):
    \[
    \lambda f. \lambda x. \text{if } (x = 0) \text{ then } 1 \text{ else } (x \ast f(x - 1))
    \]

- Then apply $\text{fix}$ to it to get a recursive function:

  \[
  \text{fix } (\lambda f. \lambda x. \text{if } (x = 0) \text{ then } 1 \text{ else } (x \ast f(x - 1)))
  \]
Encoding recursion

Some programs diverge, but can we write *useful* loops? Yes!

To write a recursive function:

▶ Write a function that takes an $f$ and calls it in place of recursion
  
  ▶ Example (in enriched language):
    
    $\lambda f. \lambda x. \text{if } (x = 0) \text{ then } 1 \text{ else } (x \times f(x - 1))$

▶ Then apply $\text{fix}$ to it to get a recursive function:
  
  ▶ $\text{fix} (\lambda f. \lambda x. \text{if } (x = 0) \text{ then } 1 \text{ else } (x \times f(x - 1)))$

$\text{fix} (\lambda f. e)$ reduces to *something* (roughly) *equivalent to* $e[(\text{fix} (\lambda f. e))/f]$, which is “unrolling the recursion once” (and further unrollings will happen as necessary).
Encoding recursion

Some programs diverge, but can we write *useful* loops? Yes!

To write a recursive function:

1. Write a function that takes an \( f \) and calls it in place of recursion
   - Example (in enriched language):
     \[
     \lambda f. \lambda x. \text{if } (x = 0) \text{ then } 1 \text{ else } (x \ast f(x - 1))
     \]
2. Then apply \( \text{fix} \) to it to get a recursive function:
   - \( \text{fix} (\lambda f. \lambda x. \text{if } (x = 0) \text{ then } 1 \text{ else } (x \ast f(x - 1))) \)

\( \text{fix} (\lambda f. e) \) reduces to *something* (roughly) *equivalent to* \( e[(\text{fix} (\lambda f. e))/f] \), which is “unrolling the recursion once” (and further unrollings will happen as necessary).

The details are intricate; the point is that you define \( \text{fix} \) only once.

- \( \text{fix}_{\text{cbv}} = \lambda f. (\lambda x. f (\lambda y. x x y))(\lambda x. f (\lambda y. x x y)) \)
- \( \text{fix}_{\text{cbn}} = \lambda f. (\lambda x. f (x x))(\lambda x. f (x x)) \)
- (technical jargon: the \( \text{Y} \) combinator(s))
Encoding numbers and arithmetic

How about numbers and arithmetic?

- Focus on natural numbers (non-negative integers), addition, is-zero, etc.

Two approaches, based on what we have so far:

- Lists of booleans for binary numbers
  - Use \texttt{fix} to implement adders, etc.
  - Just like hardware, except fixed-width avoid recursion.

- Lists (and lengths) for unary numbers
  - Zero is empty list
  - Addition is list append

Better reason: You don't need \texttt{fix}.

(Basic arithmetic is often encodable in languages where all programs terminate)

Matthew Fluet
Programming Language Theory
Lecture 07 24
Encoding numbers and arithmetic

How about numbers and arithmetic?
▶ Focus on natural numbers \((\text{non-negative integers})\), addition, is-zero, etc.

Two approaches, based on what we have so far:
▶ Lists of booleans for binary numbers
  ▶ Use \texttt{fix} to implement adders, etc.
  ▶ Just like hardware, except fixed-width avoid recursion.
▶ Lists (and lengths) for unary numbers
  ▶ Zero is empty list
  ▶ Addition is list append

But instead everybody always teaches Church numerals. Why?
▶ Tradition? Some sense of professional obligation?
▶ Better reason: You don't need \texttt{fix}.
  (Basic arithmetic is often encodable in languages where all programs terminate)
▶ Show some basics “just for fun”
Church numerals

\[
\begin{align*}
0 &= \lambda s. \lambda z. z \\
1 &= \lambda s. \lambda z. s \, z \\
2 &= \lambda s. \lambda z. s \, (s \, z) \\
3 &= \lambda s. \lambda z. s \, (s \, (s \, z))
\end{align*}
\]

Numbers encoded as two-argument functions.

The number \( n \) is represented by a function that *does something \( n \) times*.

- Composes the first argument, \( n \) times, starting with the second argument.
- Takes \( s \) (for “successor”) and \( z \) (for “zero”), composes \( s \), \( n \) times, starting with \( z \).
Church numerals

\[
\begin{align*}
0 &= \lambda s. \lambda z. z \\
1 &= \lambda s. \lambda z. s\ z \\
2 &= \lambda s. \lambda z. s\ (s\ z) \\
3 &= \lambda s. \lambda z. s\ (s\ (s\ z))
\end{align*}
\]

Numbers encoded as two-argument functions.

The number \( n \) is represented by a function that does something \( n \) times.

- Composes the first argument, \( n \) times, starting with the second argument.
- Takes \( s \) (for “successor”) and \( z \) (for “zero”), composes \( s \), \( n \) times, starting with \( z \).

To implement arithmetic operations, we cleverly pick the right lambdas for \( s \) and \( z \).
Church numerals and arithmetic

\[
0 = \lambda s. \lambda z. z \\
1 = \lambda s. \lambda z. s z \\
2 = \lambda s. \lambda z. s (s z) \\
3 = \lambda s. \lambda z. s (s (s z))
\]

\[
\text{succ} = \lambda n. \lambda s. \lambda z. s (n s z)
\]

\text{succ}: take a “number” and return a “number” that (when called) applies \textbf{s} one more time.

What \textbf{v} for \text{succ} (\text{succ} 0) \rightarrow^*_\text{cbv} \textbf{v}?
Church numerals and arithmetic

\[
\begin{align*}
0 & = \lambda s. \lambda z. z \\
1 & = \lambda s. \lambda z. s z \\
2 & = \lambda s. \lambda z. s (s z) \\
3 & = \lambda s. \lambda z. s (s (s z)) \\
\text{succ} & = \lambda n. \lambda s. \lambda z. s (n s z)
\end{align*}
\]

\text{succ}: take a “number” and return a “number” that (when called) applies \( s \) one more time.

What \( v \) for \( \text{succ} (\text{succ} 0) \rightarrow_{\text{cbv}}^* v \)? This \( v \neq 2 \), but \( v \) and 2 are equivalent.
Church numerals and arithmetic

\[
\begin{align*}
0 &= \lambda s. \lambda z. z \\
1 &= \lambda s. \lambda z. s \, z \\
2 &= \lambda s. \lambda z. s \, (s \, z) \\
3 &= \lambda s. \lambda z. s \, (s \, (s \, z)) \\
\text{succ} &= \lambda n. \lambda s. \lambda z. s \, (n \, s \, z) \\
\text{plus} &= \lambda m. \lambda n. m \, \text{succ} \, n
\end{align*}
\]

\textbf{plus:} take two “numbers” \( m \) and \( n \) and pass to \( m \) a function that yields the successor of its argument (so this will happen \( m \) times) and \( n \) (on what to start the \( m \) iterations of succession).
Church numerals and arithmetic

\[\begin{align*}
0 & = \lambda s. \lambda z. z \\
1 & = \lambda s. \lambda z. s z \\
2 & = \lambda s. \lambda z. s (s z) \\
3 & = \lambda s. \lambda z. s (s (s z)) \\
\text{succ} & = \lambda n. \lambda s. \lambda z. s (n s z) \\
\text{plus} & = \lambda m. \lambda n. m \text{ succ } n \\
\text{times} & = \lambda m. \lambda n. m (\text{plus } n) \, 0 \\
\end{align*}\]

\text{times}: \text{ take two “numbers” } m \text{ and } n \text{ and pass to } m \text{ a function that yields } n \text{ added to its argument (so this will happen } m \text{ times) and } 0 \text{ (on what to start the } m \text{ iterations of addition).}
Church numerals and arithmetic

\[\begin{align*}
0 & = \lambda s. \lambda z. z \\
1 & = \lambda s. \lambda z. s z \\
2 & = \lambda s. \lambda z. s (s z) \\
3 & = \lambda s. \lambda z. s (s (s z)) \\
\text{succ} & = \lambda n. \lambda s. \lambda z. s (n s z) \\
\text{plus} & = \lambda m. \lambda n. m \text{succ } n \\
\text{times} & = \lambda m. \lambda n. m (\text{plus } n) 0 \\
\text{isZero} & = \lambda n. n (\lambda x. \text{false}) \text{ true}
\end{align*}\]

\text{isZero:}
Church numerals and arithmetic

\[
\begin{align*}
0 & = \lambda s. \lambda z. z \\
1 & = \lambda s. \lambda z. s \, z \\
2 & = \lambda s. \lambda z. s \, (s \, z) \\
3 & = \lambda s. \lambda z. s \, (s \, (s \, z)) \\
\end{align*}
\]

\[
\begin{align*}
\text{succ} & = \lambda n. \lambda s. \lambda z. s \, (n \, s \, z) \\
\text{plus} & = \lambda m. \lambda n. m \, \text{succ} \, n \\
\text{times} & = \lambda m. \lambda n. m \, (\text{plus} \, n) \, 0 \\
\text{isZero} & = \lambda n. n \, (\lambda x. \text{false}) \, \text{true} \\
\end{align*}
\]

\[
\begin{align*}
\text{pred} & \quad \text{(with 0 sticky) the hard one; see Wikipedia or text} \\
\text{minus} & \quad \text{similar to plus, with pred instead of succ} \\
\text{areEqual} & \quad \text{subtract and test for zero}
\end{align*}
\]
Roadmap

- Motivation for a new model (done)
- CBV lambda calculus using substitution (done (almost))
- Notes on concrete syntax (done)
- Simple Lambda encodings — LC is Turing complete! (done)
- Other reduction strategies
- Defining substitution

Further ahead:

- Types, type systems, and type safety