Data + Code

Higher-order functions work well for scope and data structures

- Scope: not all memory available to all code
  - let x = 1
  - let add3 y =
    - let z = 2 in
    - x + y + z
  - let seven = add3 4

- Data: Function closures store data. Example: Association “list”
  - let cons k v lst = (fun k’ -> if k’=k then v else lst k’)
  - let lookup k lst = lst k

(Later: Objects do both too)

Adding data structures

Extending IMP with data structures isn’t too hard:

- $H ::= c \mid x \mid e \cdot e \mid e + e \mid e \cdot e \mid (e,e) \mid e.1 \mid e.2$
- $v ::= c \mid (v,v)$
- $H ::= \cdot \mid H,x \mapsto v$

Where we are

- Done: Syntax, semantics, and equivalence
  - For a language with little more than loops and global variables

- Now: Didn’t IMP leave some things out?
  - In particular: scope, functions, and data structures
  - (Not to mention threads, I/O, exceptions, strings, …)

Time for a new model…

What about functions

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

- $e ::= \ldots \mid \text{fun } x \to s$
- $v ::= \ldots \mid \text{fun } x \to s$
- $s ::= \ldots \mid e(e)$

(NO: Consider $x := 1; (\text{fun } x \to y := x)(2); \text{ans} := x$.)

Scope matters; variable name doesn’t. That is:

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

\[
H; e_1 \downarrow \text{fun } x \rightarrow s \quad H; e_2 \downarrow \nu \quad y \text{ “fresh”} \\
H; e_1(e_2) \rightarrow H; y := x; x := \nu; s; x := y
\]

- “fresh” isn’t very IMP-like but okay (think malloc)
- not a good match to how functions are implemented
- yuck: the way we want to think about something as fundamental as a call?
- NO: wrong model for most functional and OO languages
  - (Even wrong for C if \( s \) calls another function that accesses the global variable \( x \))

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 back IMP features later)

Or just borrow a model from Alonzo Church

And drop mutation, conditionals, integers (!), and loops (!)

The wrong model

\[
H; e_1 \downarrow \text{fun } x \rightarrow s \quad H; e_2 \downarrow \nu \quad y \text{ “fresh”} \\
H; e_1(e_2) \rightarrow H; y := x; x := \nu; s; x := y \\
f_1 := (\text{fun } x \rightarrow f_2 := (\text{fun } z \rightarrow \text{ans} := x + z)); \\
f_1(2); \\
x := 3; \\
f_2(4)
\]

“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:
- \( f_2(2) \) assigns to \( f_2 \) a function that adds the current value of \( x \) to its argument

The Lambda Calculus

The Lambda Calculus:

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

You apply a function by substituting the argument for the bound variable

- (There is an equivalent environment definition not unlike heap-copying; see future homework)

Example Substitutions

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

Substitution is the key operation we were missing:

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

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

A Programming Language

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

\[
e \rightarrow e'
\]

\[
e[v/x] = e' \\
(\lambda x. e) v \rightarrow e' \\
e_1 \rightarrow e'_1 \\
e_2 \rightarrow e'_2 \\
v e_2 \rightarrow v e'_2
\]

A small-step, call-by-value (CBV), left-to-right 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 \( H(x) \) in IMP because scope is what we’re interested in

This is the “heart” of functional languages like Caml

- But “real” implementations don’t substitute; they do something equivalent
Concrete-Syntax Notes

We (and Caml) 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, assume we really have ASTs (with non-leaves labeled $\lambda$ or “application”)
   - Rules may seem strange at first, but it’s the most convenient concrete syntax
   - Based on 70 years experience

Evaluation Order Matters

Careful: With CBV we need to “thunk”…

```
"if" "true" (\x. x) ((\x. x x)(\x. x x))
```

an infinite loop

diverges, but

```
"if" "true" (\x. (\z. ((\x. x x)(\x. x x)) z))
```

a value that when called diverges

doesn’t.

Encoding Pairs

The “pair ADT”:
- There is 1 constructor (taking 2 arguments) and 2 selectors
- 1st selector returns the 1st arg passed to the constructor
- 2nd selector returns the 2nd arg passed to the constructor

```
"mkpair" \x. \y. \z. x y
"fst" \p. \(\x. \y. x)
"snd" \p. \(\x. \y. y)
```

Example:
```
"snd" ("fst" ("mkpair" ("mkpair" v1 v2) v3)) \rightarrow v2
```
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 int, a float, an instruction, or a pointer?

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

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

Beware the “Turing tarpit”

Encoding Lists

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

▶ Empty list is “mkpair” “false” “false”
▶ Non-empty list is \( \lambda h. \lambda t. \text{“mkpair”} \text{“true”} (\text{“mkpair”} h t) \)
▶ Is-empty is ...
▶ Head is ...
▶ Tail is ...

Note:

▶ Not too far from how lists are implemented
▶ Taking “tail” (“tail” “empty”) will produce some lambda
▶ Just like, without page-protection hardware, null->tail->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):
    \[
    \lambda f. \lambda x. \text{if } (x = 0) \text{ then } 1 \text{ else } (x * f(x - 1))
    \]
▶ Then apply “fix” to it to get a recursive function:
  ▶ “fix” \( \lambda f. \lambda x. \text{if } (x = 0) \text{ then } 1 \text{ else } (x * f(x - 1)) \)
  ▶ “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, especially for CBV, are icky; the point is it’s possible and you define “fix” only once
▶ Not on exam: “fix” \( \lambda y. (\lambda x. g (\lambda y. x x y))(\lambda x. g (\lambda y. x x y)) \)

Encoding Arithmetic Over Natural Numbers

How about arithmetic?

▶ Focus on non-negative numbers, addition, is-zero, etc.

How I would do it based on what we have so far:

▶ Lists of booleans for binary numbers
  ▶ Zero can be the empty list
  ▶ Use fix to implement adders, etc.
  ▶ Like in hardware except fixed-width avoids recursion
▶ Or just use list length for a unary encoding
  ▶ Addition is list append

But instead everybody always teaches Church numerals. Why?

▶ Tradition? Some sense of professional obligation?
▶ Better reason: You don’t need fix: Basic arithmetic is often encodable in languages where all programs terminate
▶ In any case, we’ll show some basics “just for fun”

Church Numerals

“0” \( \lambda s. \lambda z. z \)
“1” \( \lambda s. \lambda z. s z \)
“2” \( \lambda s. \lambda z. s (s z) \)
“3” \( \lambda s. \lambda z. s (s (s z)) \)

▶ Numbers encoded with two-argument functions
▶ The “number \( i \)” composes the first argument \( i \) times, starting with the second argument
  ▶ \( z \) stands for “zero” and \( s \) for “successor” (think unary)
▶ The trick is implementing arithmetic by cleverly passing the right arguments for \( s \) and \( z \)

“successor” \( \lambda n. \lambda s. \lambda z. s (n s z) \)

successor: take “a number” and return “a number” that (when called) applies \( s \) one more time
Church Numerals

"0"  λs. λz. z
"1"  λs. λz. s z
"2"  λs. λz. s (s z)
"3"  λs. λz. s (s (s z))

"successor"  λn. λs. λz. s (n s z)
"plus"  λn. λm. λs. λz. n s (m s z)

plus: take two "numbers" and return a "number" that uses one number as the zero argument for the other

"successor"  λn. λs. λz. s (n s z)
"plus"  λn. λm. λs. λz. n s (m s z)
"times"  λn. λm. λz. n s (m s z)

Church Numerals

"0"  λs. λz. z
"1"  λs. λz. s z
"2"  λs. λz. s (s z)
"3"  λs. λz. s (s (s z))

"successor"  λn. λs. λz. s (n s z)
"plus"  λn. λm. λs. λz. n s (m s z)
"times"  λn. λm. λs. λz. n s (m s z)

times: take two "numbers" m and n and pass to m a function that adds n to its argument (so this will happen m times) and "zero" (where to start the m iterations of addition)

"successor"  λn. λs. λz. s (n s z)
"plus"  λn. λm. λs. λz. n s (m s z)
"times"  λn. λm. m ("plus" n) "zero"

isZero: an easy one, see how the two arguments will lead to the correct answer

isZero: an easy one, see how the two arguments will lead to the correct answer

Roadmap

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

Then start type systems
- Later take a break from types to consider first-class continuations and related topics