Lecture 15

CIS 341: COMPILERS

Announcements

- HW4: OAT v. 1.0
  - Parsing & basic code generation
  - **Due: March 28th**
- Midterm grades posted to gradescope later today.

Untyped lambda calculus
Substitution
Evaluation

FIRST-CLASS FUNCTIONS

Operational Semantics

- Specified using just two inference rules with judgments of the form exp $\Downarrow$ val
  - Read this notation a as "program exp evaluates to value val"
  - This is *call-by-value* semantics: function arguments are evaluated before substitution

$$\frac{}{v \Downarrow v}$$

"Values evaluate to themselves"

$$\frac{exp_1 \Downarrow (\text{fun } x \text{ -> } exp_3) \qquad exp_2 \Downarrow v \qquad exp_3\{v/x\} \Downarrow w}{exp_1\ exp_2 \Downarrow w}$$

"To evaluate function application: Evaluate the function to a value, evaluate the argument to a value, and then substitute the argument for the function. "

Variable Capture

- Note that if we try to naively "substitute" an open term, a bound variable might capture the free variables:

	(fun x -> (x y)) {(fun z -> x) / y}	Note: x is free in (fun x -> x)
=	fun x -> (x (fun z -> x))	free x is *captured!!*

- Usually *not* the desired behavior
  - This property is sometimes called "dynamic scoping"
    The meaning of "x" is determined by where it is bound dynamically, not where it is bound statically.
  - Some languages (e.g. emacs lisp) are implemented with this as a "feature"
  - But, leads to hard to debug scoping issues

Alpha Equivalence

- Note that the names of bound variables don't matter.
  - i.e. it doesn't matter which variable names you use, as long as you use them consistently

    (fun x -> y x) is the "same" as (fun z -> y z)

    the choice of "x" or "z" is arbitrary, as long as we consistently rename them
  - Two terms that differ only by consistent renaming of bound variables are called *alpha equivalent*

- The names of free variables do matter:

    (fun x -> y x) is *not* the "same" as (fun x -> z x)

  Intuitively: y an z can refer to different things from some outer scope

Fixing Substitution

- Consider the substitution operation:

$$\{e_2/x\}\ e_1$$

- To avoid capture, we define substitution to pick an alpha equivalent version of e_1 such that the bound names of e_1 don't mention the free names of e_2.
  - Then do the "naïve" substitution.

For example: (fun x -> (x y)) {(fun z -> x) / y}

= (fun x' -> (x' (fun z -> x)) *rename* x to x'

See fun.ml

IMPLEMENTING A LAMBDA CALCULUS INTERPRETER

Compiling lambda calculus to straight-line code.
Representing evaluation environments at runtime.

CLOSURE CONVERSION

Compiling First-class Functions

- To implement first-class functions on a processor, there are two problems:
  - First: we must implement substitution of free variables
  - Second: we must separate ‘code’ from ‘data’

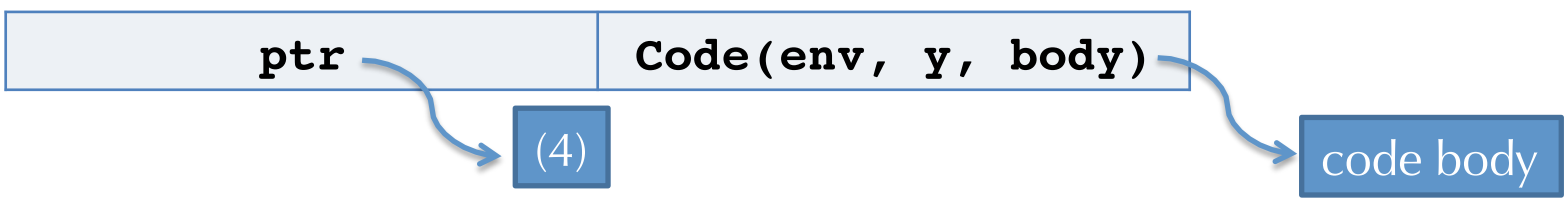
- Reify the substitution:
  - Move substitution from the meta language to the object language by making the data structure & lookup operation explicit
  - The environment-based interpreter is one step in this direction
- Closure Conversion:
  - Eliminates free variables by packaging up the needed environment in the data structure.
- Hoisting:
  - Separates code from data, pulling closed code to the top level.

Example of closure creation

- Recall the "add" function:
  `let add = fun x -> fun y -> x + y`

- Consider the inner function: `fun y -> x + y`

- When run the function application: `add 4`
  the program builds a closure and returns it.
  - The closure is a pair of the environment and a code pointer.



- The code pointer takes a pair of parameters: env and y
  - The function code is (essentially):

```
fun (env, y) -> let x = nth env 0 in x + y
```

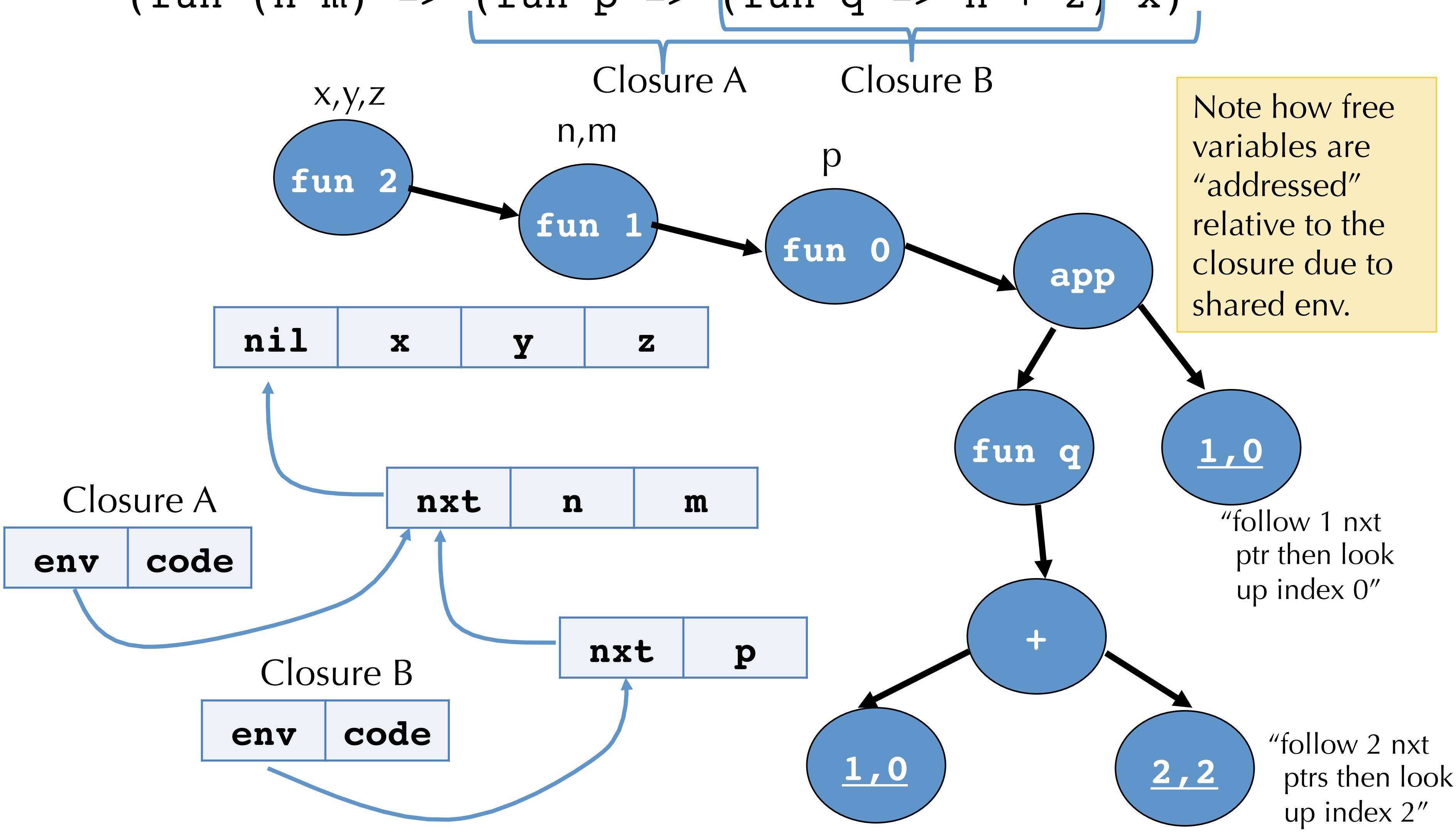
Representing Closures

- As we saw, the simple closure conversion algorithm doesn't generate very efficient code.
  - It stores all the values for variables in the environment, even if they aren't needed by the function body.
  - It copies the environment values each time a nested closure is created.
  - It uses a linked-list datastructure for tuples.

- There are many options:
  - Store only the values for free variables in the body of the closure.
  - Share subcomponents of the environment to avoid copying
  - Use vectors or arrays rather than linked structures

Array-based Closures with N-ary Functions

```
(fun (x y z) ->
   (fun (n m) -> (fun p -> (fun q -> n + z) x)
```



Note how free variables are "addressed" relative to the closure due to shared env.

Adding Integers to Lambda Calculus

exp ::=
  | …
  | n — *constant integers*
  | $exp_1 + exp_2$ — *binary arithmetic operation*

val ::=
  | `fun` x `->` exp — *functions are values*
  | n — *integers are values*

$n\{v/x\} = n$		*constants have no free vars.*
$(e_1 + e_2)\{v/x\} = (e_1\{v/x\} + e_2\{v/x\})$		*substitute everywhere*

$$\frac{exp_1 \Downarrow n_1 \quad exp2 \Downarrow n_2}{exp_1 + exp_2 \Downarrow (n1 \ [\![+]\!] \ n2)}$$

Object-level '+'

Meta-level '+'