Lecture 17

CIS 341: COMPILERS

Announcements / Plan

- HW4: OAT v. 1.0
  - Parsing & basic code generation
  - **Due: TONIGHT March 28th**

- HW5: OAT – typechecking, structs, function pointers
  - Available soon
  - Due: Thursday, April 13

- HW6: LLVM Optimization: analysis and register allocation
  - Due: Wednesday, April 26

- FINAL EXAM: Thursday, May 4th noon – 2:00p.m.

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’

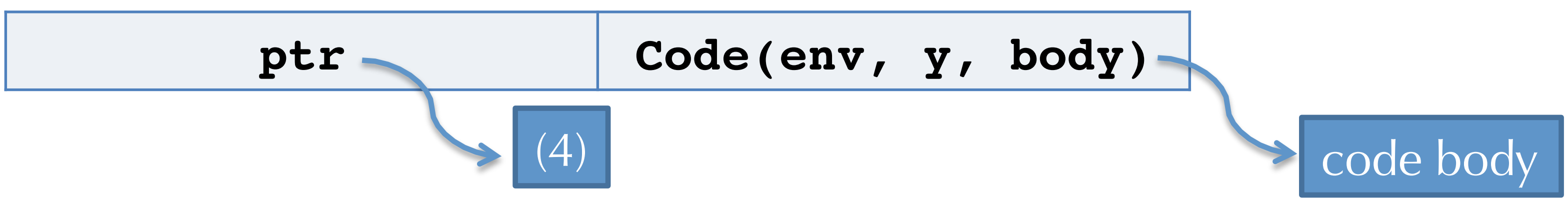
- Reify the substitution:
  - Move substitution from the meta language to the object language by making the data structure & lookup operation explicit
  - The environment-based interpreter is one step in this direction
- Closure Conversion:
  - Eliminates free variables by packaging up the needed environment 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
```

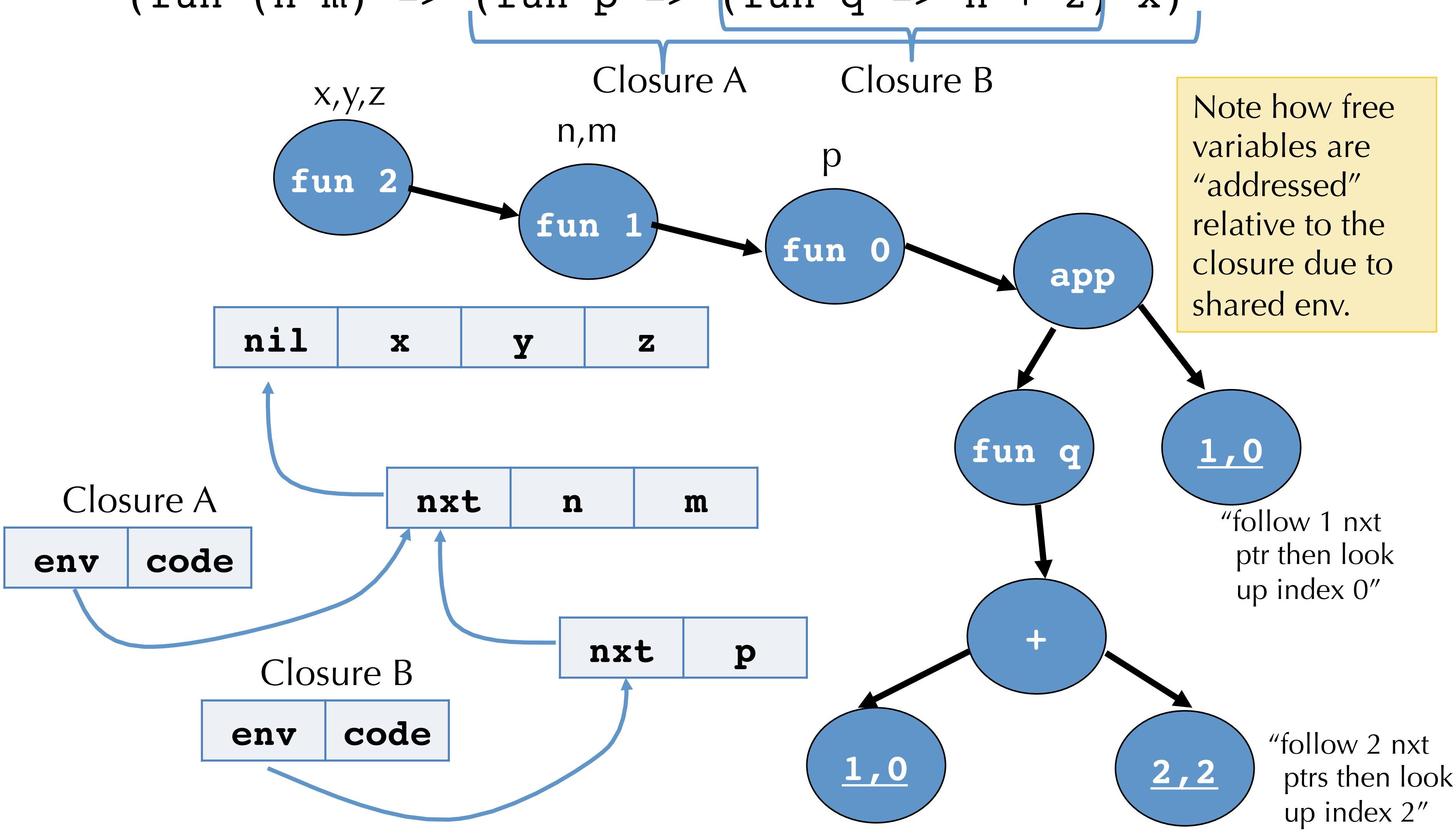
Representing Closures

- 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.

TYPECHECKING

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 '+'

NOTE: there are no rules for the case where exp1 or exp2 evaluate to functions! The semantics is *undefined* in those cases.

Type Checking / Static Analysis

- Recall the interpreter from the Eval3 module:

```
let rec eval env e =
  match e with
  | …
  | Add (e1, e2) ->
    (match (eval env e1, eval env e2) with
        | (IntV i1, IntV i2) -> IntV (i1 + i2)
        | _ -> failwith "tried to add non-integers")
  | …
```

- The interpreter might fail at runtime.
  - Not all operations are defined for all values (e.g. 3/0, 3 + true, …)
- A compiler can't generate sensible code for this case.
  - A naïve implementation might "add" an integer and a pointer

See tc.ml

STATICALLY RULING OUT PARTIALITY: TYPE CHECKING

Notes about this Typechecker

- In the interpreter, we only evaluate the body of a function when it's applied.
- In the typechecker, we always check the body of the function (even if it's never applied.)
  - We *assume* the input has some type (say t_1) and reflect this in the type of the function (t_1 **->** t_2).

- Dually, at a call site (e_1 e_2), we don't know what *closure* we're going to get.
  - But we can calculate e_1's type, check that e_2 is an argument of the right type, and also determine what type e_1 will return.

- Question: Why is this an approximation?
- Question: What if `well_typed` always returns `false`?

Type Judgments

- In the judgment: $E \vdash e : t$
  - E is a *typing environment* or a *type context*
  - E maps variables to types. It is just a set of bindings of the form: $x_1 : t_1, x_2 : t_2, \ldots, x_n : t_n$
- For example: $x : \text{int}, b : \text{bool} \vdash \text{if (b) 3 else x} : \text{int}$
- What do we need to know to decide whether "if (b) 3 else x" has type int in the environment x : int, b : bool?
  - b must be a bool i.e. $x : \text{int}, b : \text{bool} \vdash b : \text{bool}$
  - 3 must be an int i.e. $x : \text{int}, b : \text{bool} \vdash 3 : \text{int}$
  - x must be an int i.e. $x : \text{int}, b : \text{bool} \vdash x : \text{int}$

Simply-typed Lambda Calculus

- For the language in “tc.ml” we have five inference rules:

INT

$$\frac{}{E \vdash i : \text{int}}$$

VAR

$$\frac{x : T \in E}{E \vdash x : T}$$

ADD

$$\frac{E \vdash e_1 : \text{int} \qquad E \vdash e_2 : \text{int}}{E \vdash e_1 + e_2 : \text{int}}$$

FUN

$$\frac{E, x : T \vdash e : S}{E \vdash \text{fun } (x{:}T) \text{ -> } e : T \text{ -> } S}$$

APP

$$\frac{E \vdash e_1 : T \text{ -> } S \qquad E \vdash e_2 : T}{E \vdash e_1\ e_2 : S}$$

- Note how these rules correspond to the code.

Type Checking Derivations

- A *derivation* or *proof tree* has (instances of) judgments as its nodes and edges that connect premises to a conclusion according to an inference rule.
- Leaves of the tree are *axioms* (i.e. rules with no premises)
  - Example: the INT rule is an axiom
- Goal of the typechecker: verify that such a tree exists.
- Example: Find a tree for the following program using the inference rules on the previous slide:

$$\vdash \text{(fun (x:int) -> x + 3) 5} \;:\; \text{int}$$

Example Derivation Tree

$$\text{APP}\;\dfrac{\text{FUN}\;\dfrac{\text{ADD}\;\dfrac{\text{VAR}\;\dfrac{x : \text{int} \;\in\; x : \text{int}}{x : \text{int} \vdash x \;:\; \text{int}} \qquad \text{INT}\;\dfrac{}{x : \text{int} \vdash 3 \;:\; \text{int}}}{x : \text{int} \vdash x + 3 : \text{int}}}{\vdash (\text{fun } (x\text{:int}) \text{ -> } x + 3) : \text{int -> int}} \qquad \text{INT}\;\dfrac{}{\vdash 5 : \text{int}}}{\vdash (\text{fun } (x\text{:int}) \text{ -> } x + 3)\ 5 \;:\; \text{int}}$$

- Note: the OCaml function `typecheck` verifies the existence of this tree. The structure of the recursive calls when running `typecheck` is the same shape as this tree!
- Note that $x : \text{int} \in E$ is implemented by the function `lookup`

Type Safety

"Well typed programs do not go wrong."

– Robin Milner, 1978

Theorem: (simply typed lambda calculus with integers)

If $\vdash e : t$ then there exists a value v such that $e \Downarrow v$.

- Note: this is a *very* strong property.
  - Well-typed programs cannot "go wrong" by trying to execute undefined code (such as 3 + (fun x -> 2))
  - Simply-typed lambda calculus is guaranteed to terminate! (i.e. it isn't Turing complete)

Type Safety For General Languages

Theorem: (Type Safety)

If $\vdash P : t$ is a well-typed program, then either:
(a) the program terminates in a well-defined way, or
(b) the program continues computing forever

- Well-defined termination could include:
  - halting with a return value
  - raising an exception

- Type safety rules out undefined behaviors:
  - abusing "unsafe" casts: converting pointers to integers, etc.
  - treating non-code values as code (and vice-versa)
  - breaking the type abstractions of the language
- What is "defined" depends on the language semantics…