

CIS 341: Compilers

Lecture 16

The Plan

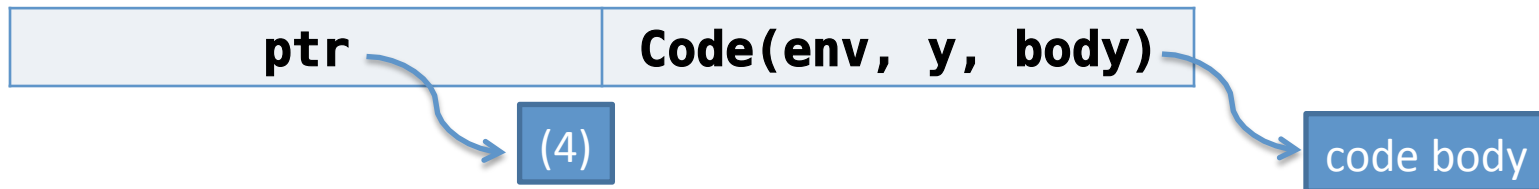
- Today:
 - Finish up closure conversion
 - Static analysis / Type checking
- Project 3: Code generation for the “T” control-transfer language
 - Due: Oct. 10th at 11:59 pm
- Announcement:
 - Midterm is Monday, Oct. 20th in class.
 - Prof. Zdancewic will be out of town

Recap: Closures & 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’
- **Closure Conversion:**
 - A closure is a pair of an environment and a code pointer.
 - Eliminates free variables by packaging up the needed context in a data structure (the environment).
- **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.



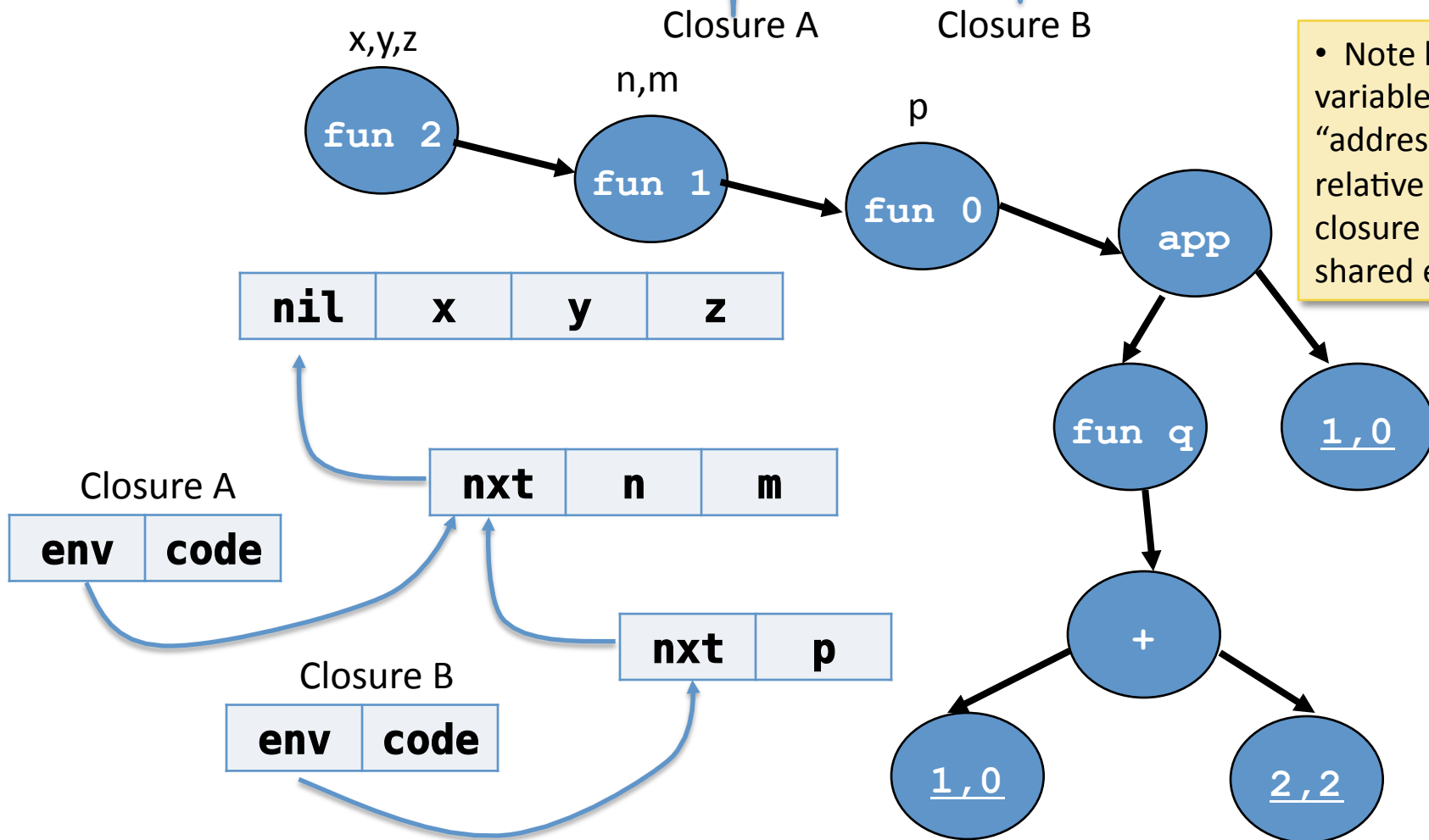
- The code pointer takes two parameters: *cenv* and *y*
 - The function code is (essentially):
fun env -> fun y -> let x = nth env 1 in x + y

Representing Closures

- The simple closure conversion algorithm in fun.ml isn't very efficient:
 - It stores all the values for variables in the environment, even if they aren't needed.
 - It copies the environment values to a new tuple each time an inner closure is created.
 - It uses a linked-list datastructure of 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 (indexing into the environment becomes more complicated)

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.

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

What to do?

- Don't worry about it... e.g. C, C++
 - Result: segmentation faults, bus errors, etc.
- Make all operations total (i.e. defined everywhere)... e.g. Scheme / Perl
 - $3 + \text{true} \rightarrow 42, \dots$ (language specifies behavior)
 - Result: unpredictable answers
- Try to rule out ill-formed programs... e.g. Java, C#, ML, Haskell
 - $3 + \text{true} \rightarrow$ compiler error:
“This expression has type bool but is here used with type int”
 - Result: predictable programs, harder to “get programs running”
- How do you know you've ruled out all ill-formed programs?

Type Soundness

- Build a model of the programming language
 - One model: an interpreter
 - Another model: constructed in mathematics
 - Usually defined via the abstract syntax
- Model defines where the language operations are partial
 - Partiality is different for different languages: e.g. “foo” + “bar” is meaningful in Java but not OCaml
- Construct a function: **well_typed : Ast.t -> bool**
 - When **well_typed e = true**, running e should not result in an undefined operation.
- Prove that the **well_typed** function is correct.
 - Such proofs are sometimes difficult, but doable for real languages (e.g. SML, Java)

Typechecking

- How do we implement the function **well_typed**?
- Big idea: “approximate” the interpreter:
 - Problem is partiality in the language semantics as defined by the interpreter.
 - Instead of interpreting the program, write a function called **typecheck** that computes a type for the program (rather than the answer obtained by running the program).
 - Behavior of **typecheck** is guided by what the interpreter would do.
- See “tc.ml”