

CIS 341: Compilers

Lecture 15

The Plan

- Today:
 - First-class functions and closure conversion
- Project 3: Code generation for the “T” control-transfer language
 - Due: Oct. 10th at 11:59 pm

“Functional” languages

- Languages like ML, Haskell, Scheme, Python, (now) C# include *first-class* functions.
- Functions can be passed as arguments (e.g. map or fold)
- Functions can be returned as values (e.g. compose)
- Functions nest: inner function can refer to variables bound in the outer function

```
let add = fun x -> fun y -> x + y
```

```
let inc = add 1
```

```
let dec = add -1
```

```
let compose = fun f -> fun g -> fun x -> f (g x)
```

```
let id = compose inc dec
```

- How do we implement such functions?

Free Variables

```
let add = fun x -> fun y -> x + y
let inc = add 1
```

- The result of **add 1** is a function
- After calling **add**, we can't throw away its argument (or local variables) because those are needed in the function returned by **add**.
- We say that the variable **x** is *free* in **fun y -> x + y**
 - Free variables are defined in an outer scope
- We say that the variable **y** is *bound* by “**fun y**” and its scope is the body “**x + y**” in the expression **fun y -> x + y**

Substitution Semantics

- Consider how to evaluate such functions:

inc = add 1

= (fun x -> fun y -> x + y) 1

= fun y -> 1 + y

- Similarly

dec = fun y -> -1 + y

- So:

id = compose inc dec

= (fun f -> (fun g -> fun x -> f (g x))) inc dec

= (fun g -> fun x -> inc (g x)) dec

= fun x -> (fun y -> 1 + y) (dec x)

= fun x -> (1 + (dec x))

= fun x -> (1 + ((fun y -> -1 + y) x))

= fun x -> (1 + (-1 + x))

Substitutions Continued

- When we *substitute* a value **v** for some variable **x** in an expression **e** (written **subst v x e**), we replace all *free occurrences* of **x** in **e** by **v**.

- Function application can be interpreted by substitution:

```
(fun x -> fun y -> x + y) 1
= subst 1 x (fun y -> x + y)
= (fun y -> 1 + y)
```

Untyped Lambda Calculus

- The lambda calculus is a minimal programming language.
 - Note: we're writing **(fun x -> e)** lambda-calculus notation: $\lambda x. e$
- It has variables, functions, and function application.
 - That's it!
 - It's Turing Complete.
 - It's the foundation for a *lot* of research in programming languages.
 - Basis for “functional” languages like Scheme, ML, Haskell, etc.

Abstract syntax in OCaml:

```
type exp =  
  | Var of var           (* local variables           *)  
  | Fun of var * exp     (* functions: fun x -> e *)  
  | App of exp * exp    (* function application *)
```

How to Implement?

- Code in fun.ml shows:
 - An substitution-based interpreter
 - Two environment-based interpreters (one broken)
 - A closure conversion translation
- 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:**
 - Eliminates free variables by packaging up the needed context in a data structure.
- **Hoisting:**
 - Separates code from data, pulling closed code to the top level.

Representing Closures

- As we saw, the simple closure conversion algorithm 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 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