Announcements

• Project 4 out
  • Due Thursday Oct 25 (14 days)
• Project 5 released today (probably)
  • Due sometime in the future
• Final exam date: Wednesday December 12, 9am
• CS Nights: Mondays 8pm, MD119
Today

- Nested functions
  - Substitution semantics
  - Environment semantics and closures
- Closure conversion
- Implementing environments and variables
  - DeBruijn indices
  - Nested environments vs flat environments
“Functional” Languages

• In functional languages, functions are first-class values
  • In addition to being called, functions can be passed as arguments (e.g., \texttt{map}), returned as results (e.g., \texttt{compose}), and stored in data structures
  • Just like other values! (\texttt{int}, \texttt{string}, ...)

• \texttt{Scheme}, \texttt{Racket}, \texttt{SML}, \texttt{OCaml}, \texttt{Haskell}, \texttt{Clojure}, \texttt{Javascript}, ...

• How do we represent a function value?

• Code pointer?
Function example

let add = fun x -> (fun y -> y+x)
let inc = add 1 (* = fun y -> y + 1 *)
let dec = add -1 (* = fun y -> y + -1 *)

let compose = fun f -> fun g -> fun x -> f(g x)
let id = compose inc dec
  (* = fun x -> inc(dec x) *)
  (* = fun x -> (fun y -> y+1)((fun y -> y-1) x) *)
  (* = fun x -> (fun y -> y+1)(x-1)) *)
  (* = fun x -> (x-1)+1 *)
Nested Functions

let add = fun x -> (fun y -> y+x)
let inc = add 1 (* = fun y -> y + 1 *)
let dec = add -1 (* = fun y -> y + -1 *)

• Consider add 1

• After calling add, we can’t throw away its argument x (or any local variables that add might use) because x is used by the nested function fun y -> y+x
Making Sense of Nested Functions

• Let’s consider what are the right semantics for nested functions
  • We will look at a simple semantics first, and then get to an equivalent semantics that we can implement efficiently
Substitution-Based Semantics

type exp = \text{Int of int | Plus of exp*exp | Var of var | Lambda of var*exp | App of exp*exp}

let rec eval (e:exp) =
match e with
| \text{Int i} -> \text{Int i}
| \text{Plus(e1,e2)} ->
  (match eval e1, eval e2 with
   | \text{Int i,Int j} -> \text{Int(i+j)})
| \text{Var x} -> error ("Unbound variable "^x)
| \text{Lambda(x,e)} -> \text{Lambda(x,e)}
| \text{App(e1,e2)} ->
  (match eval e1, eval e2 with
   (Lambda(x,e),v) ->
   eval \text{(subst v x e)})

Replace formal argument x with actual argument v
let rec subst (v:exp) (x:var) (e:exp) =
  match e with
  | Int i -> Int i
  | Plus(e1,e2) -> Plus(subst v x e1, subst v x e2)
  | Var y -> if y = x then v else Var y
  | Lambda(y,e') ->
    if y = x then Lambda(y,e')
    else Lambda(y,subst v x e')
  | App(e1,e2) -> App(subst v x e1, subst v x e2)

Slight simplification: assumes that all variable names in program are distinct.
Example

\[
((\text{fun } x \to \text{fun } y \to x + y) \ 3) \ 4
\]

\[
\text{eval App(App(Lambda(x,Lambda(y,Plus(Var x,Var y)),Int 3),Int 4)}
\]

\[
\text{eval App(Lambda(x,Lambda(y,Plus(Var x,Var y)),Int 3),Int 4)}
\]

\[
\text{eval App(Lambda(x,Lambda(y,Plus(Var x,Var y)),Int 3)}
\]

\[
\text{eval Lambda(x,Lambda(y,Plus(Var x,Var y))}
\]

\[
\text{eval Int 3}
\]

\[
\text{eval Int 4}
\]
Example

\(((\text{fun } x \to \text{fun } y \to x + y) \ 3) \ 4\)

eval \ App(\ App(\ \text{Lambda}(x,\ \text{Lambda}(y,\ \text{Plus}(\text{Var} \ x,\ \text{Var} \ y)),\ \text{Int} \ 3),\ \text{Int} \ 4))

eval \ App(\ \text{Lambda}(x,\ \text{Lambda}(y,\ \text{Plus}(\text{Var} \ x,\ \text{Var} \ y)),\ \text{Int} \ 3)) \ \text{eval} \ \text{Int} \ 4

\text{Lambda}(x,\ \text{Lambda}(y,\ \text{Plus}(\text{Var} \ x,\ \text{Var} \ y))) \ \text{Int} \ 3
Example

\[((\text{fun } x \to \text{fun } y \to x + y) \ 3) \ 4\]

eval \ App(\ App(\ \text{Lambda}(x,\ \text{Lambda}(y,\ \text{Plus}(\ \text{Var} \ x,\ \text{Var} \ y)),\ \text{Int} \ 3),\ \text{Int} \ 4)\)

eval \ App(\ \text{Lambda}(x,\ \text{Lambda}(y,\ \text{Plus}(\ \text{Var} \ x,\ \text{Var} \ y)),\ \text{Int} \ 3))\)

eval subst \ x \ (\text{Int} \ 3) \ \text{Lambda}(y,\ \text{Plus}(\ \text{Var} \ x,\ \text{Var} \ y))

eval \ \text{Lambda}(y,\ \text{Plus}(\ \text{Int} \ 3,\ \text{Var} \ y))

eval \ \text{Int} \ 4
Example

$$(\text{fun } x \rightarrow \text{fun } y \rightarrow x + y) \ 3 \ 4$$

eval \ App(\text{App}(\text{Lambda}(x,\text{Lambda}(y,\text{Plus}(\text{Var } x,\text{Var } y)),\text{Int } 3),\text{Int } 4))$

eval \ App(\text{Lambda}(x,\text{Lambda}(y,\text{Plus}(\text{Var } x,\text{Var } y)),\text{Int } 3))$

eval \ Int \ 4$

Lambda(y,\text{Plus}(\text{Int } 3,\text{Var } y))$
Example

\[
((\text{fun } x \rightarrow \text{fun } y \rightarrow x + y) \ 3) \ 4
\]

\[
\text{eval} \ \text{App}(\text{App}(\Lambda(x, \Lambda(y, \text{Plus}(\Var x, \Var y)), \text{Int } 3), \text{Int } 4)
\]

\[
\Lambda(y, \text{Plus}(\text{Int } 3, \Var y))
\]

\[
\text{eval } \text{Int } 4
\]
Example

\[ ((\text{fun } x \rightarrow \text{fun } y \rightarrow x + y) \ 3) \ 4 \]

\[
\text{eval } \text{App}(\text{App}(\text{Lambda}(x,\text{Lambda}(y,\text{Plus}(\text{Var } x,\text{Var } y)),\text{Int } 3),\text{Int } 4))
\]

\[ \text{Lambda}(y,\text{Plus}(\text{Int } 3,\text{Var } y)) \]

\[ \text{Int } 4 \]
Example

\[(\text{fun } x \rightarrow \text{fun } y \rightarrow x + y)(3)\ 4\]

eval \ App(App(Lambda(x,Lambda(y,Plus(Var x,Var y)),Int 3),Int 4))

eval subst y (Int 4) Plus(Int 3,Var y)

eval Plus(Int 3,Int 4)
Problems

- `subst` crawls over expression and replaces variable with value
- Then `eval` crawls over expression
- So `eval (subst v x e)` is not very efficient
- Why not do substitution at the same time as we do evaluation?
- Modify `eval` to use an `environment`: a map from variables to the values
First Attempt

type value = Int_v of int
type env = (string * value) list

let rec eval (e:exp) (env:env) : value =
  match e with
  | Int i -> Int_v i
  | Var x -> lookup env x
  | Lambda(x,e) -> Lambda(x,e)
  | App(e1,e2) ->
    (match eval e1 env, eval e2 env with
    | Lambda(x,e'), v -> eval e' ((x,v)::env))

• Doesn’t handle nested functions correctly!
• E.g., (fun x -> fun y -> y+x) 1 evaluates to fun y -> y+x
• Don’t have binding for x when we eventually apply this function!
Second Attempt

```
let rec eval (e:exp) (env:env) : value =
  match e with
  | Int i -> Int_v i
  | Var x -> lookup env x
  | Lambda(x,e) -> Lambda(x,subst env e)
  | App(e1,e2) ->
    (match eval e1 env, eval e2 env with
     | Lambda(x,e’), v -> eval e’ ((x,v)::env))
```

• Need to replace free variables of nested functions using environment where nested function defined

• But now we are using a version of subst again...
Closures

• Instead of doing substitution on nested functions when we reach the lambda, we can instead make a promise to finish the substitution if the nested function is ever applied.

• Instead of

\[ \text{Lambda}(x,e') \rightarrow \text{Lambda}(x,\text{subst env } e') \]

we will have, in essence,

\[ \text{Lambda}(x,e') \rightarrow \text{Promise}(\text{env}, \text{Lambda}(x, e')) \]

• Called a closure.

• Need to modify rule for application to expect environment.
**Closure-based Semantics**

```ocaml
type value = Int_v of int
         | Closure_v of {env:env, body:var*exp}
and env = (string * value) list

let rec eval (e:exp) (env:env) : value =
  match e with
  | Int i -> Int_v i
  | Var x -> lookup env x
  | Lambda(x,e) -> Closure_v{env=env, body=(x,e)}
  | App(e1,e2) ->
    (match eval e1 env, eval e2 env with
     | Closure_v{env=cenv, body=(x,e')}, v ->
       eval e' ((x,v)::cenv))
```
So, How Do We Compile Closures?

- Represent function values (i.e., closures) as a pair of function pointer and environment
- Make all functions take environment as an additional argument
  - Access variables using environment
- Can then move all function declarations to top level (i.e., no more nested functions!)
- E.g., \( \text{fun } x \rightarrow (\text{fun } y \rightarrow y+x) \) becomes, in C-like code:

```c
#include <stdio.h>
#include <stdlib.h>

typedef struct closure {
  env *env;
  int (*fn)(env *, int);
} closure;

closure *f1(env *env, int x) {
  env *e1 = extend(env, "x", x);
  closure *c = malloc(sizeof(closure));
  c->env = e1; c->fn = &f2;
  return c;
}

int f2(env *env, int y) {
  env *e1 = extend(env, "y", y);
  return lookup(e1, "y") + lookup(e1, "x");
}
```

**Closure conversion**

**Lambda lifting**
Where Do Variables Live

• Variables used in outer function may be needed for nested function
  • e.g., variable x in example on previous slide
• So variables used by nested functions can’t live on stack...
• Allocate record for all variables on heap
• Hey, this is kind of like an object!!
  • Object = struct for field values, plus pointer(s) to methods
  • Closure = environment plus pointer to code
Closure Conversion

- Converting function values into closures
  - Make all functions take explicit environment argument
  - Represent function values as pairs of environments and lambda terms
  - Access variables via environment

- E.g.,

```latex
fun x -> (fun y -> y+x)
```

becomes

```latex
fun env x ->
  let e' = extend env "x" x in
  (e', fun env y ->
    let e' = extend env "y" y in
    (lookup e' "y")+(lookup e' "x"))
```
Lambda Lifting

• After closure conversion, nested functions do not directly use variables from enclosing scope
• Can “lift” the lambda terms to top level functions!

E.g., fun env x ->

    let e' = extend env "x" x in
    (e', fun env y ->
        let e' = extend env "y" y in
        (lookup e' "y")+(lookup e' "x"))

becomes

    let f2 = fun env y ->
        let e' = extend env "y" y in
        (lookup e' "y")+(lookup e' "x")

    fun env x ->
        let e' = extend env "x" x in
        (e', f2)
Lambda Lifting

- E.g., \( \text{fun } \text{env} \ x \rightarrow \)

\[
\text{let } e' = \text{extend env } "x" \ x \text{ in } \\
(e', \ \text{fun } \text{env} \ y \rightarrow \\
\text{let } e' = \text{extend env } "y" \ y \text{ in } \\
(\text{lookup } e' "y") + (\text{lookup } e' "x"))
\]

becomes

\[
\text{let } f2 = \text{fun } \text{env} \ y \rightarrow \\
\text{let } e' = \text{extend env } "y" \ y \text{ in } \\
(\text{lookup } e' "y") + (\text{lookup } e' "x")
\]

\[
\text{fun } \text{env} \ x \rightarrow \\
\text{let } e' = \text{extend env } "x" \ x \text{ in } \\
(e', f2)
\]

closure *f1(env *env, int x) {
    env *el = extend(env,"x",x);
    closure *c =
        malloc(sizeof(closure));
    c->env = el; c->fn = &f2;
    return c;
}

int f2(env *env, int y) {
    env *el = extend(env,"y",y);
    return lookup(el, "y")
        + lookup(el, "x");
}
How Do We Compile Closures Efficiently?

• Don’t need to heap allocate all variables
  • Just the ones that “escape”, i.e., might be used by nested functions

• Implementation of environment and variables
DeBruijn Indices

• In our interpreter, we represented environments as lists of pairs of variables names and values

• Expensive string comparison when looking up variable! `lookup env x`

```ml
let rec lookup env x =
  match env with
  | ((y,v)::rest) ->
    if y = x then v else lookup rest
  | [] -> error "unbound variable"
```

• Instead of using strings to represent variables, we can use natural numbers
  • Number indicates lexical depth of variable
DeBruijn Indices

```
type exp = Int of int | Var of int
   | Lambda of exp | App of exp*exp
```

- **Original program**
  ```
  fun x -> fun y -> fun z -> x + y + z
  ```

- **Conceptually, can rename program variables**
  ```
  fun x2 -> fun x1 -> fun x0 -> x2 + x1 + x0
  ```

- **Don’t bother with variable names at all!**
  ```
  fun -> fun -> fun -> Var 2 + Var 1 + Var 0
  ```

- **Number of variable indicates lexical depth, 0 is innermost binder**
Converting to DeBruijn Indices

```ocaml
type exp = Int of int | Var of int
  | Lambda of exp | App of exp*exp

let rec cvt (e:exp) (env:var->int): D.exp =
  match e with
  | Int i -> D.Int i
  | Var x -> D.Var (env x)
  | App(e1,e2) ->
    D.App(cvt e1 env,cvt e2 env)
  | Lambda(x,e) =>
    let new_env(y) =
      if y = x then 0 else (env y)+1
    in
    Lambda(cvt e new_env)
```
New Interpreter

type value = Int_v of int
        | Closure_v of {env:env, body:exp}
and env = value list

let rec eval (e:exp) (env:env) : value =
match e with
| Int i -> Int_v i
| Var x -> List.nth env x
| Lambda e -> Closure_v{env=env, body=e}
| App(e1,e2) ->
    (match eval e1 env, eval e2 env with
     | Closure_v{env=cenv, body=(x,e’)}, v ->
       eval e’ v::cenv)
Representing Environments

((fun -> fun -> fun -> Var 2 + Var 1 + Var 0) 21) 17) 4

21

env

• Linked list (nested environments)
Representing Environments

\(((\text{fun} \to \text{fun} \to \text{fun}) \to \text{Var} \, 2 + \text{Var} \, 1 + \text{Var} \, 0) \, 21) \, 17\)

- Linked list (nested environments)
Representing Environments

\[ (((\text{fun} \rightarrow \text{fun}) \rightarrow \text{fun}) \rightarrow \text{Var}\ 2 + \text{Var}\ 1 + \text{Var}\ 0)\ 21\ 17\ 4 \]

- Linked list (nested environments)
Representing Environments

• Linked list (nested environments)
• Array (flat environment)

((fun -> fun -> fun -> Var 2 + Var 1 + Var 0) 21) 17) 4

21
17
4
env

env
Representing Environments

- Linked list (nested environments)
- Array (flat environment)
Representing Environments

- Linked list (nested environments)
- Array (flat environment)

\[((\text{fun} \to \text{fun} \to \text{fun} \to \text{Var} 2 + \text{Var} 1 + \text{Var} 0) \text{21}) \text{17}) \text{4}\]
Multiple Arguments

• Can extend DeBruijn indices to allow multiple arguments

\[
\begin{align*}
\text{fun } x \ y \ z & \rightarrow \ \text{fun } m \ n & \rightarrow \ x + z + n \\
\text{fun } & \rightarrow \ \text{fun} \rightarrow \ \text{Var}(1,0) + \ \text{Var}(1,2) + \ \text{Var}(0,1)
\end{align*}
\]

• Nested environments might then be

```text
\[
\begin{array}{cccc}
x & y & z & \text{nil} \\
m & n & \text{next}
\end{array}
\]
```
Tips For Project 5

• You will compile Scheme-like language (i.e., untyped functional language) to Cish

• Break translation down into sequence of smaller simpler passes
  • Don’t have to do entire compilation from Scish to Cish in one pass!
  • E.g., do closure conversion as one pass, then lambda lifting, then conversion to Cish, ...
  • You can define additional ASTs for intermediate passes if needed

• Be clear about what each pass is doing
  • Figure out what the invariants of each AST between passes is