Variables and Definitions

October 9, 2001

CS 131: Programming Languages

Introductory Demo

- Background facts about Emacs Lisp (Elisp)
  - Much of the functionality of Emacs (and XEmacs) editors comes from code written in LISP
    - Functions defined to do various things (like indent code)
    - Each keypress runs an associated command
  - In XEmacs, the function `mapcar*` is a predefined function which acts like `map` in rex.
    - That is, it applies a given function to the elements of a list (or the corresponding components of multiple lists)

Some Elisp Code

```
;; An example of mapcar*
(defvar succ (lambda (x) (+ x 1)))
(mapcar* succ '(1 2 3))

;; Another example of mapcar*
(let* ((y    5                    )
       (f    (lambda (x) (list x y))))
  (mapcar* f '(1 2 3)))

;; Renaming local variables
(let* ((cl-y 5                        )
       (f    (lambda (x) (list x cl-y))))
  (mapcar* f '(1 2 3)))

;; Renaming local variables again
(let* ((cl-x 5                        )
       (f    (lambda (x) (list x cl-x))))
  (mapcar* f '(1 2 3)))
```

What's going on?

An Interpreter Optimization

- Concern: applying substitutions in an interpreter is slow.

$$
\begin{array}{l}
\texttt{let } x_1 \texttt{ = 1} \\
\texttt{in let } x_2 \texttt{ = 2} \\
\quad \texttt{in let } x_3 \texttt{ = 3} \\
\quad\quad \texttt{in ...} \\
\quad\quad\quad \texttt{in let } x_n \texttt{ = } n \\
\quad\quad\quad\quad \texttt{in } x_1 + x_2 + x_3 + \ \ldots\ + x_n
\end{array}
$$

An Interpreter Optimization

- Fix: instead of replacing variables by their values (substitution) just keep a lookup table
  - This table is called an *environment*
  - I will denote an environments with ρ because it's traditional
    - (And, I already use E and e for other things)

```
let x1 = 1
in let x2 = 2
   in let x3 = 3
      in ...
         in let xn = n
            in x1+x2+x3+ ... +xn
```

Environment Interface

- Formal Semantics: Environments as finite functions

$\varnothing$	the empty environment
$\rho(x)$	the value that ρ associates with the variable x
$\rho,x{=}V$	the environment just like ρ except that it gives V as the value for x.

- ML implementation

```
type env
val empty  : env
val lookup : env * string -> absyn
val extend : env * string * absyn -> env
```

Revised Interpreter

```
exception Error

fun eval expr =
   (case expr of
       Num n => Num n
     | Bool b => Bool b
     | Nil => Nil
     | Lam _ => expr


     | Var _ => raise Error
```

```
exception Error

fun deval(env,expr) =
   (case expr of
       Num n => Num n
     | Bool b => Bool b
     | Nil => Nil
     | Lam _ => expr


     | Var v =>lookup(env,v)
```

$$\frac{}{V \Downarrow V}$$

$$\frac{}{(\rho,V) \Downarrow V}$$

$$\frac{}{(\rho,x) \Downarrow \rho(x)}$$

Revised Interpreter

```
| Plus(e1,e2) =>
  let
    val v1 = eval e1
    val v2 = eval e2
  in
    (case (v1,v2) of
       (Num m1, Num m2)
          => Num(m1+m2)
     | _  => raise Error)
  end
```

```
| Plus(e1,e2) =>
  let
    val v1 = deval (env,e1)
    val v2 = deval (env,e2)
  in
    (case (v1,v2) of
       (Num m1, Num m2)
          => Num (m1+m2)
     | _  => raise Error)
  end
```

$$\frac{M_1 \Downarrow n_1 \quad M_2 \Downarrow n_2}{M_1+M_2 \Downarrow n_1 \oplus n_2}$$

$$\frac{(\rho,M_1) \Downarrow n_1 \quad (\rho,M_2) \Downarrow n_2}{(\rho,M_1+M_2) \Downarrow n_1 \oplus n_2}$$

Revised Interpreter

```
| Equal(e1,e2) =>
   let
     val v1 = eval e1
     val v2 = eval e2
   in
     (case (v1,v2) of
        (Num m1, Num m2)
           => Bool(m1=m2)
       | _  => raise Error)
   end
```

```
| Equal(e1,e2) =>
   let
     val v1 = deval (env,e1)
     val v2 = deval (env,e2)
   in
     (case (v1,v2) of
        (Num m1, Num m2)
           => Bool(m1=m2)
       | _  => raise Error)
   end
```

$$\frac{M_1 \Downarrow n_1 \quad M_2 \Downarrow n_2}{M_1{=}{=}M_2 \Downarrow n_1{\equiv}n_2}$$

$$\frac{(\rho,M_1) \Downarrow n_1 \quad (\rho,M_2) \Downarrow n_2}{(\rho,M_1{=}{=}M_2) \Downarrow n_1{\equiv}n_2}$$

Conditional

```
| If(M,N1,N2) =>
   (case eval M of
      Bool true  => eval N1
    | Bool false => eval N2
    | _          => raise Error)
```

```
| If(M,N1,N2) =>
   (case deval(env,M) of
      Bool true  => deval(env,N1)
    | Bool false => deval(env,N2)
    | _          => raise Error)
```

$$\frac{M \Downarrow tt \quad N_1 \Downarrow v}{\text{if } M \text{ then } N_1 \text{ else } N_2 \Downarrow v}$$

$$\frac{(\rho,M) \Downarrow tt \quad (\rho,N_1) \Downarrow v}{(\rho,\text{if } M \text{ then } N_1 \text{ else } N_2) \Downarrow v}$$

$$\frac{M \Downarrow ff \quad N_2 \Downarrow v}{\text{if } M \text{ then } N_1 \text{ else } N_2 \Downarrow v}$$

$$\frac{(\rho,M) \Downarrow ff \quad (\rho,N_2) \Downarrow v}{(\rho,\text{if } M \text{ then } N_1 \text{ else } N_2) \Downarrow v}$$

Local Definitions

```
| Let(x,M,N) =>
   let
     val v1 = eval M
     val N' = subst(N,x,v1)
     val v2 = eval N'
   in
     v2
   end
```

```
| Let(x,M,N) =>
   let
     val v1 = deval(env, M)
     val env' = extend(env,x,v1)
     val v2 = deval(env', N)
   in
     v2
   end
```

$$\frac{M \Downarrow v_1 \quad N[x{\to}v_1] \Downarrow v_2}{\text{let } x{=}M \text{ in } N \Downarrow v_2}$$

$$\frac{(\rho,M) \Downarrow v_1 \quad ((\rho,x{=}v_1),N) \Downarrow v_2}{(\rho,\text{let } x{=}M \text{ in } N) \Downarrow v_2}$$

Application

```
| Apply(M,N) =>
   let
     val v1 = eval M
     val v2 = eval N
   in
     (case v1 of
        Lam(x,M') =>
           eval(subst(M',x,v2))
      | _ => raise Error)
   end
```

```
| Apply(M,N) =>
   let
     val v1 = deval(env,M)
     val v2 = deval(env,N)
   in
     (case v1 of
        Lam(x,M') =>
         let
          val env' = extend(env,x,v2)
         in
          deval(env',M')
         end
      | _ => raise Error)
   end
```

$$\frac{M \Downarrow \lambda x.M' \quad N \Downarrow v_2 \quad M'[x{\to}v_2] \Downarrow v}{M(N) \Downarrow v}$$

$$\frac{(\rho,M) \Downarrow \lambda x.M' \quad (\rho,N) \Downarrow v_2 \quad ((\rho,x{=}v_2),M') \Downarrow v}{(\rho,M(N)) \Downarrow v}$$

Pairs and Projections

```
| Pair(e1,e2) =>
     Pair(eval e1,
          eval e2)
| Fst M => (case (eval M) of
              Pair(v1,_) => v1
            | _ => raise Error)
| Snd M => (case (eval M) of
              Pair(_,v2) => v2
            | _ => raise Error))
```

```
| Pair(e1,e2) =>
     Pair(deval (env,e1),
          deval (env,e2))
| Fst M => (case deval(env,M) of
              Pair(v1,_) => v1
            | _ => raise Error)
| Snd M => (case deval(env,M) of
              Pair(_,v2) => v2
            | _ => raise Error))
```

$$\frac{M_1 \Downarrow v_1 \quad M_2 \Downarrow v_2}{<M_1,M_2> \Downarrow <v_1,v_2>} \qquad \frac{(\rho,M_1) \Downarrow v_1 \quad (\rho,M_2) \Downarrow v_2}{(\rho,<M_1,M_2>) \Downarrow <v_1,v_2>}$$

$$\frac{M \Downarrow <v_1,v_2>}{fst(M) \Downarrow v_1} \qquad \frac{M \Downarrow <v_1,v_2>}{snd(M) \Downarrow v_2} \qquad \frac{(\rho,M) \Downarrow <v_1,v_2>}{(\rho,fst(M)) \Downarrow v_1} \qquad \frac{(\rho,M) \Downarrow <v_1,v_2>}{(\rho,snd(M)) \Downarrow v_2}$$

test_input0

```
let x = 1
in let y = x+1
   in let x = y+2
      in x+y
```

What should the output be?

test_input1

```
let x = 0
in let f = λy.x+y
   in let g = λz.f(2+z)
      in g(1)
```

What should the output be?

test_input2

```
let x = 0
in let f = λy.x+y
   in let g = λz.(let q = 2
                  in f(q+z))
      in g(1)
```

What should the output be?

test_input3

```
let x = 0
in let f = λy.x+y
   in let g = λz.(let x = 2
                 in f(x+z))
      in g(1)
```

What should the output be?

Same Code in SML

```
val x = 0
fun f(y:int) = x + y
fun g(z:int) = let
                  val x = 2
               in
                  f(x + z)
               end
val _ = print (Int.toString (g 1))
```

Same Code in Emacs Lisp

```
(defvar x 0)
(defun f (y) (+ x y))
(defun g (z) (let ((x 2))
                (f (+ x z))))
(print (g 1))
```

Why the Difference?

What's going on?

```
val x = 0
fun f(y) = x + y
```

Defines f to be the function which multiplies its argument by this variable

```
fun g(z) =
    let val x = 2
    in (f (x + z))
    end
```

```
(defvar x 0)
(defun f (y) (+ x y))
```

Defines f to be the function which multiplies its argument by "x"

```
(defun g (z)
    (let ((x 2))
        (f (+ x z))
    )
)
```

More Precisely...

```
val x = 0
fun f(y) = x + y
```

f refers to the x in scope when f was *defined*. (*Static* or *Lexical* Scope)

```
fun g(z) =
    let val x = 4
    in (f z)
    end
```

```
(defvar x 0)
(defun f (y) (+ x y))
```

f refers to the most-recently defined x when f is *called*. (*Dynamic* Scope)

```
(defun g (z)
    (let ((x 4))
        (f z)
    )
)
```

Even More Precisely...

```
val x = 0
fun f(y) = x + y
```

When evaluating a call to f, free variables are looked up in the environment in place when the function was *defined*

```
fun g(z) =
    let val x = 4
    in (f z)
    end
```

```
(defvar x 0)
(defun f (y) (+ x y))
```

When evaluating a call to f, free variables are looked up in the environment in place where the function is *called*

```
(defun g (z)
    (let ((x 4))
        (f z)
    )
)
```

Scoping in Languages

- Lexical
  - Fortran, Pascal, C, C++, Java, SML, Scheme, ...
- Dynamic
  - APL, Snobol, Original LISP, Emacs LISP, Perl 4, ...
- Both
  - Perl 5, Common LISP
- What characteristics do these groups have in common?

Same Example in Perl (twice)

```
$x = 0;
sub f {
  local ($y) = @_;
  return ($x + $y);
}
sub g {
  local ($z) = @_;
  local $x = 2;
  return (f($x + $z));
}
print (g(1));
```

```
$x = 0;
sub f {
  local ($y) = @_;
  return ($x + $y);
}
sub g {
  local ($z) = @_;
  my $x = 2;
  return (f($x + $z));
}
print (g(1));
```

Interpreting Dynamic Scope

```
(defvar base 10)
(defun print_int (n)
     (… print the number n in base base …))


(let ((base 8)) (print_int 42))
(print_int 100)
```

When execution has reached this point, `base` is bound to `10` while `print_int` is bound to a function value.

Interpreting Dynamic Scope

```
(defvar base 10)
(defun print_int (n)
     (… print the number n in base base …))


(let ((base 8)) (print_int 42))
(print_int 100)
```

Here the environment has been updated to give `base` the value 8. Next the program calls `print_int`

Interpreting Dynamic Scope

```
(defvar base 10)
(defun print_int (n)
     (… print the number n in base base …))


(let ((base 8)) (print_int 42))
(print_int 100)
```

The function `print_int` looks up `base` in the environment and finds the value 8.

Interpreting Dynamic Scope

```
(defvar base 10)
(defun print_int (n)
     (… print the number n in base base …))


(let ((base 8)) (print_int 42))
(print_int 100)
```

After exiting the scope of the local variable `base`, we discard the "local" environment; `base` again refers to the global variable, which has value `10`.

Interpreting Dynamic Scope

```
(defvar base 10)
(defun print_int (n)
     (… print the number n in base base …))


(let ((base 8)) (print_int 42))
(print_int 100)
```

Thus this call to `print_int` will look up the variable `base` and find the value `10`.

Arguments for Dynamic Scope

- Easier to implement in an interpreter
- Customization of subroutines (implicit arguments)
  - What is the alternative?

```
(defvar base 10)
(defun print_int (n)
     (… print the number n in base base …))
(defun foo (y)
     (… do computation then call print_int …))

(let ((base 8)) (print_int 42))
(print_int 100)
(let ((base 2)) (foo 7))
(print_int 100)
```

Arguments for Dynamic Scoping

"Dynamic binding is especially useful for elements of the command dispatch table. For example, the RMAIL command for composing a reply to a message temporarily defines the character Control--Meta--Y to insert the text of the original message into the reply. The function which implements this command is always defined, but Control--Meta--Y does not call that function except while a reply is being edited. The reply command does this by dynamically binding the dispatch table entry for Control--Meta--Y and then calling the editor as a subroutine. When the recursive invocation of the editor returns, the text as edited by the user is sent as a reply"

Richard Stallman
EMACS: The Extensible, Customizable Display Editor

Back To Our Example

```
;; Renaming local variables again
(let* ((cl-x 5                          )
       (f    (lambda (x) (list x cl-x))))
  (mapcar* f '(1 2 3)))
```

Arguments for Lexical Scope

- Names of local variables and function arguments shouldn't matter
  - Avoids accidental clashes between separate pieces of code without having to choose obscure variable names
    - e.g., `verylongatomunlikelytobeusedbyprogrammer1`
- Easier to typecheck
  - Otherwise, what is the type of **`fn`**`(y:int)=>x*y` ?
- Easier to implement efficiently in compilers

Interpreting Static Scope

- We need a way of associating function values with the environments where they were defined.
  - The typical approach is a closure: a package that contains both the function code and information about its free variables (here, represented as an environment)

$\{\!|\lambda x.M,\rho|\!\}$ `Closure of absyn*env`

- Only two evaluation rules change:

$$\frac{}{(\rho,\lambda x.M) \Downarrow \lambda x.M}$$

$$\frac{}{(\rho,\lambda x.M) \Downarrow \{\!|\lambda x.M,\rho|\!\}}$$

$$\frac{(\rho,M) \Downarrow \lambda x.M' \quad (\rho,N) \Downarrow v_2 \quad ((\rho,x{=}v_2),M') \Downarrow v}{M(N) \Downarrow v}$$

$$\frac{(\rho,M) \Downarrow \{\!|\lambda x.M,\rho_1|\!\} \quad (\rho,N) \Downarrow v_2 \quad ((\rho_1,x{=}v_2),M') \Downarrow v}{M(N) \Downarrow v}$$

`deval -> seval`

```
| Lam(x,M) => Lam(x,M)
| Apply(M,N) =>
  let
    val v1 = deval(env,M)
    val v2 = deval(env,N)
  in
   (case v1 of
     Lam(x,M') =>
      let
       val env'=extend(env,x,v2)
      in
       deval(env',M')
      end
    | _ => raise Error)
 end
| Closure(M,env) => raise Error
```

```
| Lam(x,M) => Closure(Lam(x,M),env)
| Apply(M,N) =>
  let
    val v1 = seval(env,M)
    val v2 = seval(env,N)
  in
   (case v1 of
     Closure(Lam(x,M'),env1) =>
      let
       val env'=extend(env1,x,v2)
      in
       seval(env',M')
      end
    | _ => raise Error)
 end
| Closure(M,env) => Closure(M,env)
```