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

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

;; Another example of mapcar*
defun succ (lambda (x) (+ x 1))
(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.

```lisp
let x_1 = 1
in let x_2 = 2
    in let x_3 = 3
    in ...
    in let x_n = n
    in x_1 + x_2 + x_3 + ... + x_n
```
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 \( \rho \) because it's traditional
    - (And, I already use \( E \) and \( e \) for other things)

```
let x_1 = 1
  in let x_2 = 2
      in let x_3 = 3
      in ...
      in let x_n = n
          in x_1 + x_2 + x_3 + ... + x_n
```

Environment Interface

- Formal Semantics: Environments as finite functions
  - the empty environment
  - \( \rho(x) \) the value that \( \rho \) associates with the variable \( x \)
  - \( \rho, x = V \) the environment just like \( \rho \) except that it gives \( V \) as the value for \( x \).

- ML implementation

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

Revised Interpreter

```ml
exception Error

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

val v = lookup(env, v)
```

Revised Interpreter

```ml
fun deval(env, expr) =
  (case expr of
    Num n => Num n
    | Bool b => Bool b
    | Nil => Nil
    | Lam _ => expr
    | Var v => lookup(env, v)
    | 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)
```

\[ M_1 \Downarrow n_1 \quad M_2 \Downarrow n_2 \]
\[ (\rho, M_1) \Downarrow n_1 \quad (\rho, M_2) \Downarrow n_2 \]
\[ (\rho, M_1 + M_2) \Downarrow n_1 \oplus n_2 \]
Revised Interpreter

<table>
<thead>
<tr>
<th>Equal(e1,e2) =&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>let val v1 = eval e1 val v2 = eval e2 in (case (v1,v2) of (Num m1, Num m2) =&gt; Bool(m1=m2) end)</td>
</tr>
</tbody>
</table>

Local Definitions

<table>
<thead>
<tr>
<th>Let(x,M,N) =&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>let val v1 = eval M val v2 = subst(N,x,v1) in v2 end</td>
</tr>
</tbody>
</table>

Application

<table>
<thead>
<tr>
<th>Apply(M,N) =&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>let val v1 = eval M val v2 = eval N in (case v1 of Lam(x,M') =&gt; eval(subst(M',x,v2)) end)</td>
</tr>
</tbody>
</table>

Conditional

<table>
<thead>
<tr>
<th>If(M,N1,N2) =&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>(case eval M of Bool true =&gt; eval N1</td>
</tr>
</tbody>
</table>

M ⊢ tt N₁ ⊨ v
(p,M) ⊢ tt (p,N₁) ⊨ v
(p,let x=M in N) ⊨ 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) |

\[
\begin{align*}
M &\Downarrow v₁, M &\Downarrow v₂ \\
\langle M₁, M₂ \rangle &\Downarrow <v₁,v₂>
\end{align*}
\]

\[
\begin{align*}
M &\Downarrow <v₁,v₂> \\
\text{fst}(M) &\Downarrow v₁ \\
\text{snd}(M) &\Downarrow v₂
\end{align*}
\]

\[
\begin{align*}
(\rho, M₁) &\Downarrow v₁, (\rho, M₂) &\Downarrow v₂ \\
(\rho, \langle M₁, M₂ \rangle) &\Downarrow <v₁,v₂>
\end{align*}
\]

\[
\begin{align*}
(\rho, M) &\Downarrow <v₁,v₂> \\
(\rho, \text{fst}(M)) &\Downarrow v₁ \\
(\rho, \text{snd}(M)) &\Downarrow v₂
\end{align*}
\]

---

**test_input0**

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

What should the output be?

---

**test_input1**

```latex
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**

```latex
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 = \(y \cdot x + y\) 
in let g = \(z \cdot (\text{let } x = 2 \\text{ in } f(x+z))\) 
in g(1)

What should the output be?

Same Code in SML

```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

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

Why the Difference?
What's going on?

\[
\begin{align*}
\text{val } x &= 0 \\
\text{fun } f(y) &= x + y
\end{align*}
\]

Defines \( f \) to be the function which multiplies its argument by this variable.

\[
\begin{align*}
\text{fun } g(z) &= \\
&\quad \text{let val } x = 2 \\
&\quad \text{in } (f (x + z))
\end{align*}
\]

More Precisely...

\[
\begin{align*}
\text{val } x &= 0 \\
\text{fun } f(y) &= x + y
\end{align*}
\]

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

\[
\begin{align*}
\text{fun } g(z) &= \\
&\quad \text{let val } x = 4 \\
&\quad \text{in } (f z)
\end{align*}
\]

\[
\begin{align*}
\text{val } x &= 0 \\
\text{fun } f(y) &= x + y
\end{align*}
\]

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

\[
\begin{align*}
\text{fun } g(z) &= \\
&\quad \text{let (x 4)} \\
&\quad \text{in } (f z)
\end{align*}
\]

Even More Precisely...

\[
\begin{align*}
\text{val } x &= 0 \\
\text{fun } f(y) &= x + y
\end{align*}
\]

When evaluating a call to \( f \), free variables are looked up in the environment in place where the function was defined.

\[
\begin{align*}
\text{fun } g(z) &= \\
&\quad \text{let val } x = 4 \\
&\quad \text{in } (f z)
\end{align*}
\]

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)

```perl
$x = 0;
sub f {
    local ($y) = @_; 
    return ($x + $y);
}

sub g {
    local ($z) = @_; 
    local $x = 2;
    return (f($x + $z));
}

print (g(1));
```

Interpreting Dynamic Scope

```lisp
(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.

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)

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.

Arguments for Dynamic Scope

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

(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)

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

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
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., verylongatomunliketobeusedbyprogrammer1
- Easier to typecheck
  - Otherwise, what is the type of \( \text{fn}(y:\text{int}) \Rightarrow x*y \) ?
- 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:

\[
\begin{align*}
(\rho, \lambda x. M) & \Downarrow \lambda x. M \\
(\rho, \lambda x. M) & \Downarrow [\lambda x. M, \rho] \\
(\rho, M) & \Downarrow \lambda x. M^* (\rho, N) \Downarrow v_1 \\
& \Downarrow \mathcal{M}(N) \Downarrow v
\end{align*}
\]

\[
\begin{align*}
(\rho, M) & \Downarrow \lambda x. M^* (\rho, N) \Downarrow v_1 \\
& \Downarrow \mathcal{M}(N) \Downarrow v
\end{align*}
\]

\[
\begin{align*}
(\rho, M) & \Downarrow \lambda x. M^* (\rho, N) \Downarrow v_1 \\
& \Downarrow \mathcal{M}(N) \Downarrow v
\end{align*}
\]

\[
\begin{align*}
(\rho, M) & \Downarrow \lambda x. M^* (\rho, N) \Downarrow v_1 \\
& \Downarrow \mathcal{M}(N) \Downarrow v
\end{align*}
\]

\[
\begin{align*}
(\rho, M) & \Downarrow \lambda x. M^* (\rho, N) \Downarrow v_1 \\
& \Downarrow \mathcal{M}(N) \Downarrow v
\end{align*}
\]

\[
\begin{align*}
(\rho, M) & \Downarrow \lambda x. M^* (\rho, N) \Downarrow v_1 \\
& \Downarrow \mathcal{M}(N) \Downarrow v
\end{align*}
\]

\[
\begin{align*}
(\rho, M) & \Downarrow \lambda x. M^* (\rho, N) \Downarrow v_1 \\
& \Downarrow \mathcal{M}(N) \Downarrow v
\end{align*}
\]

deval \rightarrow seval

\[
\begin{align*}
\text{Lam}(s, M) & \Rightarrow \text{Lam}(s, M) \\
\text{Apply}(M, N) & \Rightarrow
\text{let}
\text{val} v_1 = \text{deval}(\mathcal{E}, M)
\text{val} v_2 = \text{deval}(\mathcal{E}, N)
in
\text{case} v_1 \text{ of}
\text{Lam}(x, M') \Rightarrow
\text{let}
\text{val} \mathcal{E}' = \text{extend}(\mathcal{E}, x, v_2)
in
\text{deval}(\mathcal{E}', M')
\text{end}
\Rightarrow \text{raise Error}
\text{end}
\text{Val}(M, \mathcal{E}) \Rightarrow \text{raise Error}
\end{align*}
\]

\[
\begin{align*}
\text{Lam}(s, M) & \Rightarrow \text{Closure}(\text{Lam}(s, M), \mathcal{E}) \\
\text{Apply}(M, N) & \Rightarrow
\text{let}
\text{val} v_1 = \text{seval}(\mathcal{E}, M)
\text{val} v_2 = \text{seval}(\mathcal{E}, N)
in
\text{case} v_1 \text{ of}
\text{Closure}(\text{Lam}(s, M'), \mathcal{E}_1) \Rightarrow
\text{let}
\text{val} \mathcal{E}' = \text{extend}(\mathcal{E}_1, x, v_2)
in
\text{seval}(\mathcal{E}', M')
\text{end}
\Rightarrow \text{raise Error}
\text{end}
\text{Val}(M, \mathcal{E}) \Rightarrow \text{Closure}(M, \mathcal{E})
\end{align*}
\]