Static and Dynamic Scope

February 18, 2002

CS 131: Programming Languages

Where We've Been

- Parsing takes in concrete syntax and yields abstract syntax trees (AST's)
- Last week we looked at an interpreter (the function `eval`) for a very simple functional language.
  - The language was defined by giving a formal semantics defining an evaluation relation

    $$exp \Downarrow value$$

  - The interpreter agreed with the semantics:

    `eval(`*exp*`)=`*value* iff *exp* $\Downarrow$ *value*

Assignment 4

- Concern: applying substitutions in an interpreter is slow.

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

Interpreter Optimization

- Fix: instead of eagerly replacing variables by their values (substitution) just keep a lookup table
  - This table is called an *environment*

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

A Problem?

- The optimized interpreter will treat the following two programs differently.

```
let x be 1
in let addx be (y)=>y+x
   in let z be 4
      in addx(z)
```

```
let x be 1
in let addx be (y)=>y+x
   in let x be 4
      in addx(x)
```

Assignment 4 Summary

- The semantics in class and in the assignment define two different languages!
  - with the same abstract syntax
- Is one of the two languages "wrong" ?

Example Abstract Syntax

```
let x = 0
in let f = ((y) => x+y)
   in let g be ((z)=> let x be 2
                      in f(x+z) )
      in g(1)
```

What will the output be for `eval`, `evalEnv`?

Equivalent SML Code

```
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))
```

What's going on?

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

Defines f to be the function which adds its argument to the value of 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 adds its argument to "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)
   ))
```

Scoping in Languages

- Static Scoping
  - Fortran, Pascal, C, C++, Java, SML, Scheme, ...
- Dynamic Scoping
  - APL, Snobol, (Original) LISP, Emacs LISP, Perl 4, ...
- Both
  - Perl 5, Common LISP

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));
```

An Argument for Dynamic Scope

- Customization of subroutines (implicit arguments)

```
(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)
```

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`.

Another Argument

"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

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?

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

- It's not enough to remember the code (source code or compiled machine instructions) of a function
- We need a way of remembering information about the free variables of the function when it was defined.

```
let x = 0
in let f = ((y) => x+y)
   in let g be ((z)=> let x be 2
                       in f(x+z) )
      in g(1)
```

Interpreting Static Scope

- Common implementation: instead of just passing the function code around, pass around a *closure*
  - Package containing code + any information required about the function's *free* variables
- Here it can just be the function + *the environment when the function was defined*
  - Informal abstract syntax: ⦃(*var*)=>*exp*,env⦄
  - SML representation:

```
datatype absyn = ...
               | Closure of absyn*absyn env
```

Change 1

- Evaluation of function values

$$\frac{}{(\mathtt{env},(\mathit{var})\texttt{=>}\mathit{exp}) \Downarrow (\mathit{var})\texttt{=>}\mathit{exp}}$$

$$\downarrow$$

$$\frac{}{(\mathtt{env},(\mathit{var})\texttt{=>}\mathit{exp}) \Downarrow [\![(\mathit{var})\texttt{=>}\mathit{exp},\mathtt{env}]\!]}$$

Change 2

- Evaluation of Applications

$$\frac{\begin{array}{ll} (\mathit{env},\mathit{exp}_1) & \Downarrow (\mathit{var})\texttt{=>}\mathtt{exp}_3 \\ (\mathit{env},\mathit{exp}_2) & \Downarrow \mathit{value}_2 \\ ((\mathit{env},\mathtt{var}=\mathit{value}_2),\mathit{exp}_3) & \Downarrow \mathit{value}_3 \end{array}}{(\mathtt{env},\mathtt{exp}_1\ \mathtt{exp}_2) \quad \Downarrow \mathit{value}_3}$$

$$\downarrow$$

$$\frac{\begin{array}{ll} (\mathit{env},\mathit{exp}_1) & \Downarrow [\![(\mathit{var})\texttt{=>}\mathit{exp}_3,\mathit{env}_1]\!] \\ (\mathit{env},\mathit{exp}_2) & \Downarrow \mathit{value}_2 \\ ((\mathit{env}_1,\mathtt{var}=\mathit{value}_2),\mathit{exp}_3) & \Downarrow \mathit{value}_3 \end{array}}{(\mathtt{env},\mathtt{exp}_1\ \mathtt{exp}_2) \quad \Downarrow \mathit{value}_3}$$

evalEnv -> evalStatic

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

Interpreting Static Scope

- What's in the environment at each point?

```
let x = 0
in let f = ((y) => x+y)
   in let g be ((z)=> let x be 2
                       in f(x+z) )
      in g(1)
```