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
    \[ \text{exp} \Downarrow \text{value} \]
  - The interpreter agreed with the semantics:
    \[ \text{eval} (\text{exp}) = \text{value} \text{ iff } \text{exp} \Downarrow \text{value} \]

Assignment 4

- Concern: applying substitutions in an interpreter is slow.

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

Interpreter Optimization

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

```
let x_1 be 1
in let x_2 be 2
  in let x_3 be 3
  in ...
  in let x_n be n
  in x_1 + x_2 + x_3 + ... + x_n
```
A Problem?

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

```sml
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

```sml
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

```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

```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?

```lisp
(val x 0)
(fun f (y) (+ x y))
Defines x 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)
```

More Precisely...

```lisp
(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)
```

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)

```perl
$x = 0;
sub f {
    local ($y) = @_;  # Customization of subroutines (implicit arguments)
    return ($x + $y);
}
sub g {
    local ($z) = @_;  # Customization of subroutines (implicit arguments)
    local $x = 2;
    return (f($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.

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

<table>
<thead>
<tr>
<th>Arguments for Lexical Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>- Names of local variables and function arguments shouldn’t matter</td>
</tr>
<tr>
<td>- Avoids accidental clashes between separate pieces of code without having to choose obscure variable names</td>
</tr>
<tr>
<td>- e.g., verylongatomunlikelytobeusedbyprogrammer1</td>
</tr>
<tr>
<td>- Easier to typecheck</td>
</tr>
<tr>
<td>- Otherwise, what is the type of $\text{fn}(y: \text{int}) \Rightarrow x \ast y$?</td>
</tr>
<tr>
<td>- Easier to implement efficiently in compilers</td>
</tr>
</tbody>
</table>

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: $[(\text{var}) \Rightarrow \text{exp}, \text{env}]$
• SML representation:
  
  ```sml
datatype absyn = ...
  | Closure of absyn*absyn env
  ```
Change 1

- Evaluation of function values

\[(\text{env}, (\text{var}) \Rightarrow \text{exp}) \cup (\text{var}) \Rightarrow \text{exp}]\]

\[(\text{env}, (\text{var}) \Rightarrow \text{exp}) \cup \{(\text{var}) \Rightarrow \text{exp}, \text{env}\}\]

evalEnv \rightarrow \text{evalStatic}

\[
\begin{align*}
| \text{Lam}(x,M) &\Rightarrow \text{Lam}(x,M) \\
| \text{Apply}(M,N) &\Rightarrow \\
&\text{let} \\
&\quad \text{val v1 = evalEnv(env,M)} \\
&\quad \text{val v2 = evalEnv(env,N)} \\
&\text{in} \\
&\quad \text{(case v1 of} \\
&\quad \quad \text{Lam}(x,M') \Rightarrow \\
&\quad \quad \quad \text{let} \\
&\quad \quad \quad \quad \text{val env' = extend(env,v1,v2)} \\
&\quad \quad \quad \quad \text{in} \\
&\quad \quad \quad \quad \text{evalEnv(env',M')} \\
&\quad \quad \text{end} \\
&\quad \quad \text{| _ => raise Error} \\
&\text{end} \\
&\text{| Closure(M,env) => raise Error}
\end{align*}
\]

\[
\begin{align*}
| \text{Lam}(x,M) &\Rightarrow \text{Closure(Lam(x,M),env)} \\
| \text{Apply}(M,N) &\Rightarrow \\
&\text{let} \\
&\quad \text{val v1 = evalStatic(env,M)} \\
&\quad \text{val v2 = evalStatic(env,N)} \\
&\text{in} \\
&\quad \text{(case v1 of} \\
&\quad \quad \text{Closure(Lam(x,M'),env1) \Rightarrow} \\
&\quad \quad \quad \text{let} \\
&\quad \quad \quad \quad \text{val env' = extend(env1,v1,v2)} \\
&\quad \quad \quad \quad \text{in} \\
&\quad \quad \quad \quad \text{evalStatic(env',M')} \\
&\quad \quad \text{end} \\
&\quad \quad \text{| _ => raise Error} \\
&\text{end} \\
&\text{| Closure(M,env) => Closure(M,env)}
\end{align*}
\]

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)