Scribe: Filip Radliński Lecturer: Radu Rugina

1 Prelim Review

Half the lecture consisted of a brief prelim review, a full solution will be made available separately.

2 Review of Previous Lecture

Recall the language F, which uses a number of constructs from IMP, REC and λ-calculus. These include: arithmetic, λ-calculus and recursive functions, conditional operators, pairs, and the let construct. We use this language as a vehicle for exploring the semantics of other language constructs. In the operational semantics of F don't deal with errors explicitly, but rather get into a stuck state where we cannot continue. In the denotational semantics we use domain equations and the ecase construct to deal with errors as well as non-termination explicitly. Recall the syntax of the ecase construct:

$$\begin{array}{rl} \textit{ecase}\ \ e\ \textit{of} & \textit{in}(x_1) \Rightarrow e_1 \\ | & \textit{in}(x_2) \Rightarrow e_2 \\ & \vdots \\ | & \textit{in}(x_i) \Rightarrow e_i \end{array}$$

If e does not terminate or returns an error state, the whole ecase expression does not terminate or returns an error. Also, for the default case (that is if e doesn't match any pattern) the construct returns an error.

3 Scope of Variables

We next explore the semantics of *scoping*, which refers to the binding of variable occurrences to their declarations or definitions. We are all familiar with scoping in C and Java, where each variable occurrence is bound to a declaration of that variable in an enclosing block or to a declaration of that variable as an argument of an enclosing procedure. In the case of nested blocks, a variable declaration in a block is visible in all of the nested sub-blocks.

Scoping for F is similar, but instead of declarations within blocks, we have declarations for let constructs. For instance, a construct let $x = 3$ in e declares a variable e whose scope is the expression e. Therefore, each occurrence of x in e is bound to the declaration $x = 3$. Again, we may have nested let constructs, which may yield a hierarchy of scopes.

However, scoping becomes more problematic for procedures. The difficulties occur for variables used in the procedure body, but declared outside the procedure. The problem is that there are multiple possibilities for binding these variables to their declarations: we could use the declarations enclosing the definition of the procedure; or we could use the declarations enclosing each invocation of the procedure. These two alternatives correspond to the two possible kinds of scoping:

- *Static scoping:* we bound variable occurrences to their declarations enclosing the procedure definition. Therefore, all function calls will use the same declarations of variables.
- *Dynamic scoping:* we bound variable occurrences to their declarations enclosing each procedure invocation. Hence, different procedure calls may use different declarations.

This is best illustrated with an example. Consider the following piece of F code:

```
let x = 1 in
   let foo = λ y. y + x in
      ( let x = 2 in foo(0) ) +
      ( let x = 3 in foo(0) )
```

The question that arises is how we bind the x in foo when we invoke it, because we have a different let-bindings of x for the function definition and for each of the invocations. With static scoping, we use the declaration let x=1, always. So foo(0) = 0+1 both times. Hence the program returns $(0+1)+(0+1)=2$. With dynamic scoping, we use the binding let x=2 at the first call and the binding let x=3 at the second. Hence the program returns $(0+2)+(0+3)=2+3=5$.

4 Denotational Semantics for Scoping

Denotational semantics allow us to precisely express which kind of scoping we wish to use. We previously had the following, which I claim to be the denotational semantics for static scoping:

$$\begin{array}{rcl} \mathcal{C}[\![\lambda x.\, e]\!]\rho & = & \lambda v.\, \mathcal{C}[\![e]\!]\rho[x \to v] \\ \mathcal{C}[\![e_0 e_1]\!]\rho & = & \mathit{ecase}\ \ \mathcal{C}[\![e_0]\!]\rho\ \mathit{of}\ \ \mathit{Function}(f)\ \Rightarrow \\ & & \mathit{ecase}\ \ \mathcal{C}[\![e_1]\!]\rho\ \mathit{of}\ \ \mathit{Value}(v) \Rightarrow f\ v \end{array}$$

Here, variable occurrences in the body of an abstraction are always bound to the declarations enclosing the abstraction. These declarations correspond to the environment ρ in the first equation, which is the *static*, or *lexical* environment of the function.

Let's now look at the semantics of dynamic scoping. What we want is to evaluate an abstraction without using the environment enclosing the abstraction. We instead evaluate the abstraction to a function which takes the environment as a parameter; we then decide dynamically, at each call site, what this environment should be. We therefore extend the domain of functions to include the dynamic environment as a parameter:

$$\mathit{Function} = \mathit{Value} \to \mathit{Env} \to \mathit{Result}$$

We can now write the denotational semantics for dynamic scoping as follows:

$$\begin{array}{rcl} \mathcal{C}[\![\lambda x.\, e]\!]\rho_{stat} & = & \lambda v.\, \lambda\rho_{dyn}.\, \mathcal{C}[\![e]\!]\rho_{dyn}[x \to v] \\ \mathcal{C}[\![e_0 e_1]\!]\rho_{dyn} & = & \mathit{ecase}\ \ \mathcal{C}[\![e_0]\!]\rho_{dyn}\ \mathit{of}\ \ \mathit{Function}(f) \Rightarrow \\ & & \mathit{ecase}\ \ \mathcal{C}[\![e_1]\!]\rho_{dyn}\ \mathit{of}\ \ \mathit{Value}(v) \Rightarrow f\ v\ \rho_{dyn} \end{array}$$

Note that in the first equation, the static environment ρ_{stat} (enclosing the function definition) is ignored. The result of the evaluation produces a function that takes the dynamic environment as parameter. The second equation shows that each application uses the dynamic environment at the call site.

5 Comments and Questions

- We modify our example code to be the following:

```
let x = 1 in
   let foo = λ y. y + x in
      ( let x = 2 in foo(0) + x) +
      ( let x = 3 in foo(0) + x)
```

  In this example, the bold occurrences of **x** after the calls to foo are not bound to the environment enclosing the function definition, regardless of what kind of scoping we use. The first occurrence of bold **x** uses the binding let x = 2; the second occurrence of bold **x** uses let x = 3. This is because **x** occurs outside the function body, and scoping only refers to the variables in the function body.

- For non-recursive functions, dynamic scoping is effectively equivalent to inlining functions. If we inline foo at each of the two call sites, we get the same behavior as the original program with dynamic scoping.

- Consider the original program and eliminate the first line "let x = 1 in". With static scoping, this will result in an error; with dynamic scoping there will be no error in the execution of the program.

- For recursive functions, dynamic scoping can model the behavior of the rec construct. Consider the following recursive function definition:

$$\begin{array}{l}\mathsf{let\ f = rec\ y\ (\lambda x.\ if\ (x = 1)\ then\ y\ else\ y(x-1))} \\ \mathsf{in\ \ f(3)}\end{array}$$

  We get the same result for the following program if we use dynamic scoping:

$$\begin{array}{l}\mathsf{let\ f = \lambda x.\,if\ (x = 1)\ then\ f\ else\ f(x-1)} \\ \mathsf{in\ \ f(3)}\end{array}$$

  See the posting on the newsgroup for a detailed explanation.