CS611 Lecture 26 **Static and Dynamic Scoping** 3 November 2003

Scribe: Ruijie Wang Lecturer: Radu Rugina

Last class, we talked about the denotational semantics of FUN and how it handles errors. We defined recursive domains and used the *ecase* construct to match values in the domains. Today we will review some interesting rules we presented in the end of last class and use an example to illustrate how it works.

1 Rules Review

$$\begin{aligned}
\mathcal{C}[\![e_0\ e_1]\!]\rho &= \mathit{ecase}\ \mathcal{C}[\![e_0]\!]\rho\ \mathit{of}\ \mathit{Function}(f) \Rightarrow \\
&\quad\ \mathit{ecase}\ \mathcal{C}[\![e1]\!]\rho\ \mathit{of}\ \mathit{Value}(v) \Rightarrow f\ v \\
\mathcal{C}[\![\lambda x.e]\!]\rho &= \lambda v \in \mathit{Value}\,.\,\mathcal{C}[\![e]\!]\ \rho\ [x \to v] \\
\mathcal{C}[\![\mathsf{rec}\ y\ (\lambda x.\,e)]\!]\rho &= \mathit{fix}(\lambda f \in \mathit{Function}\,.\,\mathcal{C}[\![\lambda x.e]\!]\rho[y \to f])
\end{aligned}$$

An interesting question is why not use $\mathit{fix}(\lambda y\ .\ \mathcal{C}[\![\lambda x.e]\!]\rho)$ here. Note that y is a piece of syntax. We do not want syntax to appear in the mathematical objects that the denotational semantics constructs. In $\mathit{fix}(\lambda f \in \mathit{Function}\,.\,\mathcal{C}[\![\lambda x.e]\!]\rho[y \to f])$, f is not a piece of syntax, it is in the *Function* domain.

2 Example

Let's use an example to see how these rules work.

$$\mathcal{C}[\![(\lambda x.1)\,\overbrace{(\mathsf{rec}\ y\ (\lambda x.\,yx))}^{A}]\!]\rho_0$$

Here, $\mathsf{rec}\ y\ (\lambda x.\,yx)$ is a recursive function that calls itself. If written in C, it would be:

```
int f(int x) {
    return f(x);
}
```

The execution of this function would not terminate. But note that this is only an abstraction: the evaluation of the abstraction itself terminates; but when it is applied to an argument, the evaluation does not terminate.

$$\begin{aligned}
\mathcal{C}[\![(\lambda x.1)(\mathsf{rec}\ y\ (\lambda x.\,yx))]\!]\rho_0 &= \mathit{ecase}\ \mathcal{C}[\![\lambda x.1]\!]\rho_0\ \mathit{of}\ \mathit{Function}(f) \Rightarrow \\
&\quad\ \mathit{ecase}\ \mathcal{C}[\![A]\!]\rho_0\ \mathit{of}\ \mathit{Value}(v') \Rightarrow f\ v' \\
&= \mathit{ecase}\ \mathcal{C}[\![A]\!]\rho_0\ \mathit{of}\ \mathit{Value}(v') \Rightarrow \lambda v.1\ v' \\
&\quad\ \ (\text{Since } \mathcal{C}[\![\lambda x.1]\!]\rho_0 = \lambda v.\mathcal{C}[\![1]\!]\rho[x \to v] = \lambda v.1) \\
&= (\lambda v.1)(\lambda v.\bot)\ =\ 1
\end{aligned}$$

The final step in this evaluation follows from:

$$\begin{aligned}
\mathcal{C}[\![\mathsf{rec}\ y\ (\lambda x.\,yx)]\!]\rho_0 &= \mathit{fix}(\lambda f.\mathcal{C}[\![\lambda x.yx]\!]\rho_0[y \to f]) \\
&= \mathit{fix}(\lambda f.\lambda v.\mathcal{C}[\![yx]\!]\rho_0[y \to f][x \to v]) \\
&= \mathit{fix}(\lambda f.\lambda v.fv) \\
&= \lambda v.\bot
\end{aligned}$$

And the final step in that evaluation holds because, when computing a fixed point, we start with $\lambda v.\bot$, repeatedly apply the function $\lambda f.\lambda v.fv$, and $(\lambda f.\lambda v.fv)(\lambda v.\bot) = \lambda v.\bot$.

Recall that we said when $\mathsf{rec}\, y\ (\lambda x.\, yx)$ is applied to an argument, the evaluation does not terminate. Consider how the evaluation of the above program would change if we add an argument to it:

$$\mathcal{C}[\![(\lambda x.1)((\mathsf{rec}\, y\ (\lambda x.\, yx))\ 2)]\!]\rho_0$$

Will we get $(\lambda v.1)(\perp)$?

The answer is no. Our denotational semantics use the *ecase* construct:

$$\mathit{ecase}\, \mathcal{C}[\![A]\!]\rho_0 \ldots$$

If $\mathcal{C}[\![A]\!]\rho_0$ evaluates to $\perp$, then the whole expression evaluates to $\perp$.

3 Scoping

Now we discuss the semantics of scoping, which refers to the binding of variable occurrences to their declarations or definitions.

In FUN, we use the let construct to express scoping:

$$\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2$$

Here, e_2 is the scope of x. All free occurrences of x in e_2 are bound to e_1. But not *all* occurrences of x in e_2 are bound to e_1, since there might be other let constructs inside e_2 which bind x to other values.

When we discuss the scoping for functions, we have two choices. We can bind a variable occurrence inside a function to the declaration enclosing the definition of the function or to the declaration enclosing the calls to the function:

- Static scoping (lexical scoping): we bind the variable inside a function to the declaration enclosing the definition of the function.
- Dynamic scoping: we bind the variable inside a function to the declaration enclosing the calls to the function.

Example:

$$\begin{array}{l} \mathsf{let}\ x = 1\ \mathsf{in} \\ \quad \mathsf{let}\ \mathit{foo} = \lambda y.y + x\ \mathsf{in} \\ \quad\quad (\mathsf{let}\ x = 2\ \mathsf{in}\ \mathit{foo}(0)) + \\ \quad\quad (\mathsf{let}\ x = 3\ \mathsf{in}\ \mathit{foo}(0)) \end{array}$$

During the first call of *foo*, if we use static scoping, the result of *foo* would be $\mathit{foo} = \lambda v.v + 1$. If we use dynamic scoping, the semantics of *foo* would be $\mathit{foo} = \lambda v.v + 2$

Our current semantics for FUN supports static scoping. Of the programming languages we commonly use, LaTeX and Perl support dynamic scoping. C and Java also have a form of dynamic scoping in the case of global variables (in C) and static fields (in Java).

4 Denotational Semantics for Dynamic Scoping

We want to evaluate an abstraction without using the lexical environment enclosing the abstraction. We instead evaluate the abstraction to a function which takes the environment as a parameter. Then during each call, we decide dynamically what this environment should be. So we extend the domain of functions to include the dynamic environment as a parameter:

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

And we rewrite some of the denotational semantics:

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

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

5 Example of Using Dynamic Scoping

We now use an example to show how these rules work.

$$\begin{array}{rl} & \mathcal{C}[\![\mathsf{let}\ x = 1\ \mathsf{in}\ \mathsf{let}\ \mathit{foo} = \lambda y.y + x\ \mathsf{in}\ \mathsf{let}\ x = 2\ \mathsf{in}\ \mathit{foo}\ 0]\!]\rho_0 \\ \\ = & \mathit{ecase}\ \mathcal{C}[\![1]\!]\rho_0\ \mathit{of\ Value}(v) \Rightarrow \mathcal{C}[\![\mathsf{let}\ \mathit{foo} = \lambda y.y + x\ \mathsf{in}\ \mathsf{let}\ x = 2\ \mathsf{in}\ \mathit{foo}\ 0]\!]\rho_0[x \to v] \\ \\ = & \mathcal{C}[\![\mathsf{let}\ \mathit{foo} = \lambda y.y + x\ \mathsf{in}\ \mathsf{let}\ x = 2\ \mathsf{in}\ \mathit{foo}\ 0)]\!]\rho_1 \qquad (\text{Here, } \rho_1 = \rho_0[x \to 1]) \\ \\ = & \mathit{ecase}\ \mathcal{C}[\![\lambda y.y + x]\!]\rho_1\ \mathit{of\ Value}(v) \Rightarrow \mathcal{C}[\![\mathsf{let}\ x = 2\ \mathsf{in}\ \mathit{foo}\ 0]\!]\rho_1[\mathit{foo} \to v] \\ \\ = & \mathcal{C}[\![\mathsf{let}\ x = 2\ \mathsf{in}\ \mathit{foo}\ 0]\!]\rho_2 \quad \left(\begin{array}{l} \text{Here,} \\ \begin{array}{rcl} \rho_2 & = & \rho_1[\mathit{foo} \to f] \\ f & = & \mathcal{C}[\![\lambda y.y + x]\!]\rho_1 \\ & = & \lambda v \in \mathit{Value}\,.\,\lambda\rho \in \mathit{Env}\,.\,\mathcal{C}[\![y + x]\!]\rho[y \to v] \\ & = & \lambda v\,.\,\lambda\rho\,.\,\mathit{ecase}\ \mathcal{C}[\![y]\!]\rho[y \to v]\ \mathit{of}\ \mathbb{Z}(n_1) \Rightarrow \\ & & \quad \mathit{ecase}\ \mathcal{C}[\![x]\!]\rho[y \to v]\ \mathit{of}\ \mathbb{Z}(n_2) \Rightarrow n_1 + n_2 \end{array} \end{array} \right) \\ \\ = & \mathit{ecase}\ \mathcal{C}[\![2]\!]\rho_2\ \mathit{of\ Value}(v) \Rightarrow \mathcal{C}[\![\mathit{foo}\ 0]\!]\rho_2[x \to v] \\ \\ = & \mathcal{C}[\![\mathit{foo}\ 0]\!]\rho_3 \qquad (\text{Here, } \rho_3 = \rho_2[x \to 2]) \\ \\ = & \mathit{ecase}\ \mathcal{C}[\![\mathit{foo}]\!]\ \rho_3\ \mathit{of\ Function}(f) \Rightarrow \\ & \quad \mathit{ecase}\ \mathcal{C}[\![0]\!]\ \rho_3\ \mathit{of\ Value}(v) \Rightarrow f\ v\ \rho_3 \\ \\ = & f\ 0\ \rho_3 \\ \\ = & \mathit{ecase}\ 0\ \mathit{of}\ \mathbb{Z}(n_1) \Rightarrow \\ & \quad \mathit{ecase}\ \mathcal{C}[\![x]\!]\rho_3[y \to 0]\ \mathit{of}\ \mathbb{Z}(n_2) \Rightarrow n_1 + n_2 \\ \\ = & 0 + 2 \\ \\ = & 2 \end{array}$$