1 Introduction

In the last two lectures, we studied the static and dynamic approaches to variable scoping. These scoping disciplines are mechanisms for binding names to values so that the values can later be retrieved by their assigned name.

Both static and dynamic naming strategies are *hierarchical* in the sense that variables enter and leave scope according to the program's abstract syntax tree (in the case of static scoping) or the tree of function calls (in dynamic scoping). This dependence on hierarchy might prove restrictive or inflexible for writing certain kinds of programs. In such cases we might want a more liberal naming discipline that is independent of any hierarchy induced by the syntactic structure or call structure.

Such a non-hierarchical scoping structure is provided by *modules*. A *module* is like a software black box with its own local namespace. It can export resources as a set of names without revealing its internal composition. Thus the names that can be used at a certain point in a program need not "come down from above" but can also be names exported by modules.

Good programming practices encourage *modularity*, especially in the construction of large systems. Programs should be composed of discrete components that communicate with one another along small, well-defined interfaces and are reusable. Modules are consistent with this idea. Each module can treat the others as black boxes; that is, they know nothing about what is inside the other module except as revealed by the interface.

Early programming languages had one global namespace in which names of all functions in source files and libraries were visible to all parts of the program. This was the approach for example of FORTRAN and C. There are certain problems with this:

- The various parts of the program can become tightly coupled. In other words, the global namespace does not enforce the modularity of the program. Replacing any particular part of the program with an enhanced equivalent can require a lot of effort.
- Undesired name collisions occur frequently, since names inevitably tend to coincide.
- In very large programs, it is difficult to come up with unique names, thus names tend to become non-mnemonic and hard to remember.

A solution to this problem is for the language to provide a *module mechanism* that allows related functions, values, and types to be grouped together into a common context. This allows programmers to create a local namespace, thus minimizing naming conflicts. Examples of modules are classes in Java and C++, packages in Java, or structures in ML. Given a module, we can access the variables in it by qualifying the variable names with the name of the module, or we can *import* the whole namespace of the module into our code, so we can use the module's names as if they had been declared locally.

2 Modules

A *module* is a collection of named things (such as values, functions, types etc.) that are somehow related to one another. The programmer can choose which names are *public* (exported to other parts of the program) and which are *private* (inaccessible outside the module).

There are usually two ways to access the contents of a module. The first is with the use of a *selector expression*, where the name of the module is prefixed to the variable in a certain way. For instance, we write `m.x` in Java and `m::x` in C++ to refer to the entity with name `x` in the module `m`.

The second method of accessing the contents of a module is to use an expression that brings names from a module into scope for a section of code. For example, we can write something like `with m do e`, which means that `x` can be used in the block of code `e` without prefixing it with `m`. In ML, for instance, the

command "Open List" brings names from the module List into scope. In C++ we write "using namespace module_name;" and in Java we write "import module_name;" for the similar purposes.

Another issue is whether to have modules as *first class* or *second class* objects. First class objects are entities that can be passed to a function as an argument, bound to a variable or returned from a function. In ML, modules are not first class objects, whereas in Java, modules can be treated as first class objects using the *reflection mechanism*. While first-class treatment of modules increases the flexibility of a language, it usually requires some extra overhead at run-time.

3 Module Syntax and Translation to uML

We now extend uML, our simple ML-like language, to support modules. We call the new language uML+M to denote that it supports modules. There must be some values that we can use as names with an equality test. The syntax of the new language is:

$$
\begin{array}{lcll}
e & ::= & \ldots & \\
 & | & \mathsf{module}\ (x_1 = e_1, \ldots, x_n = e_n) & \text{(module definition)} \\
 & | & e_m.e & \text{(selector expression)} \\
 & | & \mathsf{with}\ e_m\ e & \text{(bringing into scope)} \\
 & | & \mathsf{lookup-error} & \text{(error)}
\end{array}
$$

We now want to define a translation from uML+M to uML.[1] To do this, we notice that a module is really just an environment, since it is a mapping from names to values. Here is a translation of the module definition:

$$
\begin{array}{l}
[\![\mathsf{module}\ (x_1 = e_1, x_2 = e_2, \ldots, x_n = e_n)]\!]\ \rho \stackrel{\triangle}{=} \\
\quad \lambda x.\ \mathsf{if}\ x = x_1\ \mathsf{then}\ [\![e_1]\!]\ \rho\ \mathsf{else} \\
\qquad \mathsf{if}\ x = x_2\ \mathsf{then}\ [\![e_2]\!]\ \rho\ \mathsf{else} \\
\qquad \ldots \\
\qquad \mathsf{if}\ x = x_n\ \mathsf{then}\ [\![e_n]\!]\ \rho\ \mathsf{else} \\
\qquad \mathsf{lookup\text{-}error}
\end{array}
$$

The above is one possible translation. Note that ρ is passed as the environment to the translation of $e_1, \ldots, e_n$. This has an important consequence: variables defined within the module are *not* visible within the initialization expression of other variables in the module. For instance, in the above translation, we cannot refer to any of the x_i's within any of the e_i's. Nor does it seem possible to use the resulting environment within itself for the purpose of accessing the module variables, since this leads to circularity problems.

However, we could translate module using techniques from the translation of letrec. The environment that results after the translation should be the same that is used within the module. This can be found by taking the fixpoint of the function

$$
\begin{array}{l}
\lambda\rho'.\ \lambda x.\ \mathsf{if}\ x = x_1\ \mathsf{then}\ [\![e_1]\!]\ \rho'\ \mathsf{else} \\
\qquad \mathsf{if}\ x = x_2\ \mathsf{then}\ [\![e_2]\!]\ \rho'\ \mathsf{else} \\
\qquad \ldots \\
\qquad \mathsf{if}\ x = x_n\ \mathsf{then}\ [\![e_n]\!]\ \rho'\ \mathsf{else} \\
\qquad \mathsf{lookup\text{-}error}
\end{array}
$$

[1] Actually our translation is to the target language uML+S+lookup-error, a simple extension to uML that contains equality operators for strings and an extra token called lookup-error, which is returned when a variable name is not found in a module.

This works fine if the e_i's are functions. However, it is not clear how to handle variables whose initialization expressions reference one another. For instance, consider

$$\mathsf{module}\ (x_1 = x_2,\ x_2 = x_1)$$

Java and C++ avoid this problem by allowing functions within a class to call any other function within the class, but initialization expressions of variables are only allowed to refer to variables declared earlier in the class. For our purposes, we stay with the translation above.

The remainder of the translation is as follows:

$$\begin{array}{rcl} [\![e_m.e]\!]\ \rho & \triangleq & ([\![e_m]\!]\ \rho)\ ([\![e]\!]\ \rho) \\ [\![\mathsf{with}\ e_m\ e]\!]\ \rho & \triangleq & [\![e]\!]\ (\mathsf{MERGE}\ \rho\ ([\![e_m]\!]\ \rho)) \\ \mathsf{MERGE}\ \rho\ \rho' & \triangleq & \lambda x.\,\mathsf{let}\ y = \rho'\ x\ \mathsf{in}\ \mathsf{if}\ y = \mathsf{lookup\text{-}error}\ \mathsf{then}\ \rho\ x\ \mathsf{else}\ y \end{array}$$

In the translation of $[\![e_m.e]\!]$, presumably $[\![e]\!]\ \rho$ would evaluate to a name.

Note that these translations are really functions of environments. That is, the translation above for $[\![e_m.e]\!]$ can be written:

$$[\![e_m.e]\!] \quad = \quad \lambda\rho.\,([\![e_m]\!]\ \rho)\ ([\![e]\!]\ \rho).$$

4 State

Program state refers to the ability to change the values of program variables over time. The λ-calculus and the uML language do not have state in the sense that once a variable is bound to a value, it is impossible to change that value as long as the variable is in scope. Although state is not a necessary feature of a programming language—for example, the λ-calculus is Turing complete but does not have a notion of state—it is a common feature of most languages, and most programmers are accustomed to it.

4.1 Reference Cells

We extend uML to include the ability to change the values of variables, and we call the new language uML!. We use a construct called a *reference cell* (aka *mutable variable*, aka *pointer variable*) that allows the variable to be rebound to different values over time. The syntax of uML! is as follows:

e	::=	...	
	|	$\mathsf{ref}\ e$	(create a reference cell with initial value e)
	|	$!e$	(evaluation or dereference)
	|	$e_1\ :=\ e_2$	(assignment or mutation)
	|	unit	(no value)

Informally, the meaning of these new expressions is as follows. Evaluating the expression $\mathsf{ref}\ e$ creates a new reference cell having the initial value e. The expression $!e$ evaluates to the current value of the reference cell e. The assignment $e_1 := e_2$ assigns the value of e_2 to the reference cell e_1, and in our semantics returns the value unit.

Other design choices could have been made. For example, the assignment operator could have returned the value of e_2 so that expressions like $e_0\ :=\ (e_1\ :=\ e_2)$ would make sense and could be used to assign the same value to multiple variables.

In this treatment, reference cells are first-class objects. This is more powerful than simple mutable variables as in C. It is also more dangerous, because it introduces *aliasing*, as the following example demonstrates:

$$
\begin{array}{l}
\mathsf{let}\ x = \mathsf{ref}\ 1\ \mathsf{in} \\
\quad \mathsf{let}\ y = x\ \mathsf{in} \\
\quad\quad \mathsf{let}\ z = (x := 2)\ \mathsf{in} \\
\quad\quad\quad !y
\end{array}
$$

Here y is an alias for x. Dereferencing y gives us the value of x, which in this case has changed from 1 to 2. The expression therefore evaluates to 2. In other words, if you kick x, y jumps!

4.2 Operational Semantics

We now give an operational semantics for reference cell expressions in uML!. To do this, we define a *configuration* as a pair $\langle e, \sigma \rangle$, where e is an expression and σ is a function mapping reference cells to values. Reference cells are denoted generically by ℓ.

$$
\frac{e \underset{\text{uML}}{\longrightarrow} e'}{\langle e, \sigma \rangle \underset{\text{uML!}}{\longrightarrow} \langle e', \sigma \rangle}
$$

$$
\begin{aligned}
\langle \mathsf{ref}\ v, \sigma \rangle &\ \rightarrow\ \langle \ell, \sigma[v/\ell] \rangle, \quad \ell \notin \operatorname{dom} \sigma \\
\langle !\ell, \sigma \rangle &\ \rightarrow\ \langle \sigma(\ell), \sigma \rangle, \quad \ell \in \operatorname{dom} \sigma \\
\langle \ell := v, \sigma \rangle &\ \rightarrow\ \langle \mathsf{unit}, \sigma[v/\ell] \rangle, \quad \ell \in \operatorname{dom} \sigma
\end{aligned}
$$

where $\sigma[v/\ell]$ is the same function as σ, except that the value on input ℓ is changed to v. These are eager evaluation rules, evaluating values from left to right, so the evaluation contexts are given by the grammar:

$$
E \quad ::= \quad [\bullet] \ \mid\ \mathsf{ref}\ E \ \mid\ !E \ \mid\ E := e \ \mid\ \ell := E.
$$