1 Introduction

We have 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.

The scope of a variable is determined by the program's abstract syntax tree in the case of static scoping and the tree of function calls in dynamic scoping. Both strategies impose restrictions on the scope that may prove inflexible for writing certain kinds of programs.

A more liberal naming discipline is provided by *modules*. A *module* is like a software black box with its own local namespace. It can export resources by name 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 practice encourages *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 lets programmers create local namespaces, thereby minimizing naming conflicts. Examples of modules are classes in Java and C++, packages in Java, or structures in OCaml. 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. 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 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 a name x declared in m can be used in the block of code e without prefixing it with m. In OCaml, 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 OCaml, 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 runtime.

3 Module Syntax and Translation to FL

We now extend FL, our simple functional language, to support modules. We call the new language FL-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:

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

We now want to define a translation from FL-M to FL. To do this, we notice that a module is really just an environment, since it is a mapping from names to values. Thus we will use the technology for environments developed in the last lecture. We will also want the definitions of the x_i to be mutually recursive, so we will use letrec. Here is a translation, where ρ_0 is a representation of the empty environment:

$$
\begin{array}{rcl}
[\![\mathsf{module}\ (x_1 = e_1, \ldots, x_n = e_n)]\!]\,\rho & \triangleq & \mathsf{letrec}\ x_1 = [\![e_1]\!]\,\rho\ \mathsf{and}\ \cdots\ \mathsf{and}\ x_n = [\![e_n]\!]\,\rho\ \mathsf{in} \\
 & & \mathsf{let}\ \rho_1 = \mathsf{update}\ \rho_0\ x_1\ \text{“}x_1\text{”}\ \mathsf{in} \\
 & & \mathsf{let}\ \rho_2 = \mathsf{update}\ \rho_1\ x_2\ \text{“}x_2\text{”}\ \mathsf{in} \\
 & & \ldots \\
 & & \mathsf{let}\ \rho_n = \mathsf{update}\ \rho_{n-1}\ x_n\ \text{“}x_n\text{”}\ \mathsf{in} \\
 & & \rho_n \\
 & & \\
[\![e_m.x]\!]\,\rho & \triangleq & \mathsf{lookup}\ ([\![e_m]\!]\,\rho)\ \text{“}x\text{”} \\
[\![\mathsf{with}\ e_m\ e]\!]\,\rho & \triangleq & [\![e]\!]\ (\mathsf{merge}\ ([\![e_m]\!]\,\rho)\ \rho) \\
\mathsf{merge}\ \rho\ \rho' & \triangleq & \lambda x.\,\mathsf{let}\ y = \mathsf{lookup}\ \rho\ x\ \mathsf{in} \\
 & & \quad \mathsf{if}\ y \neq \mathsf{error}\ \mathsf{then}\ y \\
 & & \quad \mathsf{else}\ \mathsf{lookup}\ \rho'\ x
\end{array}
$$

4 State

Program state refers to the ability to change the values of program variables over time. The λ-calculus and the FL 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 Programming Paradigms

Two major programming paradigms are *functional* (stateless) and *imperative* (stateful). In a purely functional language, expressions resemble mathematical formulas. This allows the programmer to reason equationally, avoiding many of the pitfalls associated with a constantly changing execution environment. For example, in a functional language, it is always the case that

$$x = e \quad \Leftrightarrow \quad f(x) = f(e).$$

Concurrency is easier to implement with a functional language because of confluence (aka the Church–Rosser property).

On the other hand, imperative programming more closely resembles the way we perceive the real world in that there exists an underlying notion of *state* that can change over time. We have seen an example of state and imperative programming with the language IMP.

5 Mutable Variables

Mutable variables (aka *pointers*, aka *references*) provide another level of mutable state. Mutable variables can be updated in a way that cannot be handled by the simple substitution rules of their functional counterparts. They are somewhat more complicated than ordinary variable bindings because they introduce the extra complication of *aliasing*—the possibility of naming the same data value with different names.

For example, consider the following code:

```
let x = ref 1 in
let y = x in
let z = (x := 2) in
!y
```

The first x points to a newly allocated location holding the value 1. Then y is assigned x, the pointer to the location holding 1. Then the value pointed to by x is updated to be 2. When y is dereferenced with $!y$, the result is now 2. Here x and y are aliases of the same data value. When you kick x, y jumps!

6 The FL! Language

6.1 Syntax

The syntax for FL! is as follows. There is a countable set Loc of *memory locations*, denoted generically by ℓ, that can hold data values. All FL expressions are FL! expressions. In addition, there are a few more:

$$e \quad ::= \quad \ldots \;\mid\; \mathsf{ref}\, e \;\mid\; !e \;\mid\; e_1 := e_2 \;\mid\; e_1 \,;\, e_2 \;\mid\; \ell$$

6.2 The Store

We define a *store* as a partial function $\sigma : \mathsf{Loc} \rightharpoonup \mathsf{Val}$ with finite domain. A store is very much like an environment, except that now variables are bound to locations, not to the data values themselves, and the

locations are bound to data values. We use the following functions to manipulate stores:

$$\begin{array}{rcl}
\mathsf{lookup}\ \sigma\ \ell & = & \sigma(\ell) \\
\mathsf{update}\ \sigma\ \ell\ v & = & \sigma[v/\ell] \\
\mathsf{malloc}\ \sigma\ v & = & (\ell,\, \sigma[v/\ell]) \quad \text{where } \ell \text{ is a new location not already in } \mathsf{dom}\,\sigma \\
\mathsf{empty} & = & \text{the completely undefined store with domain } \varnothing.
\end{array}$$

Here $\sigma[v/\ell]$ refers to the store σ with the location ℓ changed to contain the value v, if $\ell \in \mathsf{dom}\,\sigma$, otherwise it refers to σ with the new location ℓ containing value v added to $\mathsf{dom}\,\sigma$.

6.3 Small-Step Semantics

A program in FL! is a configuration $\langle e,\, \sigma\rangle$, where e is an FL! expression and σ is a store. The small-step SOS is given by augmenting FL with the following additional evaluation contexts and reduction rules:

$$E \quad ::= \quad \ldots \ \mid\ \mathsf{ref}\,E \ \mid\ !E \ \mid\ E := e \ \mid\ v := E \ \mid\ E\,;\,e$$

The hole $[\cdot]$ is already included in the $\ldots$. The evaluation contexts generated by the above grammar are all the contexts $E[\cdot]$ in which a reduction may be applied. The contexts specify a family of rules collectively called the *context rule*

$$\frac{\langle e,\, \sigma\rangle \to \langle e',\, \sigma'\rangle}{\langle E[e],\, \sigma\rangle \to \langle E[e'],\, \sigma'\rangle}$$

The reduction rules are

$$\begin{array}{cc}
\langle \mathsf{ref}\ v,\, \sigma\rangle \to \langle \ell,\, \sigma[v/\ell]\rangle,\ \ell \notin \mathsf{dom}\,\sigma & \langle !\ell,\, \sigma\rangle \to \langle \sigma(\ell),\, \sigma\rangle,\ \ell \in \mathsf{dom}\,\sigma \\
\langle \ell := v,\, \sigma\rangle \to \langle \mathsf{null},\, \sigma[v/\ell]\rangle,\ \ell \in \mathsf{dom}\,\sigma & \langle v; e,\, \sigma\rangle \to \langle e,\, \sigma\rangle.
\end{array}$$

It can be shown by induction that it is impossible to create dangling pointers in FL!.

7 Translating FL! to FL

The following translation maps an FL! expression e to a function $[\![e]\!]$ taking an environment ρ and store σ and producing a pair $\langle e',\, \sigma'\rangle$, where e' is an FL expression and σ' is a store. Here $\mathsf{let}\ \langle b,\, \sigma'\rangle = [\![e_0]\!]\rho\sigma\ \mathsf{in}\ \ldots$ is syntactic sugar for

$$\mathsf{let}\ p = [\![e_0]\!]\rho\sigma\ \mathsf{in}\ \mathsf{let}\ b = \#1\ p\ \mathsf{in}\ \mathsf{let}\ \sigma' = \#2\ p\ \mathsf{in}\ \ldots\ .$$

$$\begin{array}{rcl}
[\![n]\!]\rho\sigma & = & \langle n,\, \sigma\rangle \\
[\![x]\!]\rho\sigma & = & \langle \rho(x),\, \sigma\rangle \\
[\![\mathsf{if}\ e_0\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2]\!]\rho\sigma & = & \mathsf{let}\ \langle b,\, \sigma'\rangle = [\![e_0]\!]\rho\sigma\ \mathsf{in} \\
 & & \mathsf{if}\ b\ \mathsf{then}\ [\![e_1]\!]\rho\sigma'\ \mathsf{else}\ [\![e_2]\!]\rho\sigma' \\
[\![\mathsf{ref}\ e]\!]\rho\sigma & = & \mathsf{let}\ \langle x,\, \sigma'\rangle = [\![e]\!]\rho\sigma\ \mathsf{in}\ \mathsf{malloc}\,\sigma'\,x \\
[\![!e]\!]\rho\sigma & = & \mathsf{let}\ \langle x,\, \sigma'\rangle = [\![e]\!]\rho\sigma\ \mathsf{in}\ \langle \mathsf{lookup}\,\sigma'\,x,\, \sigma'\rangle \\
[\![e_1 := e_2]\!]\rho\sigma & = & \mathsf{let}\ \langle x_1,\, \sigma_1\rangle = [\![e_1]\!]\rho\sigma\ \mathsf{in} \\
 & & \mathsf{let}\ \langle x_2,\, \sigma_2\rangle = [\![e_2]\!]\rho\sigma_1\ \mathsf{in} \\
 & & \langle \mathsf{null},\, \mathrm{update}\ \sigma_2\ x_1\ x_2\rangle \\
[\![e_1\,;\,e_2]\!]\rho\sigma & = & \mathsf{let}\ \langle x,\, \sigma_1\rangle = [\![e_1]\!]\rho\sigma\ \mathsf{in}\ [\![e_2]\!]\rho\sigma_1 \\
[\![\lambda x.\, e]\!]\rho_{\mathrm{lex}}\sigma_{\mathrm{lex}} & = & \lambda v\sigma_{\mathrm{dyn}}.\, [\![e]\!]\rho_{\mathrm{lex}}[v/y]\sigma_{\mathrm{dyn}} \\
[\![e_1\ e_2]\!]\rho_{\mathrm{dyn}}\sigma_{\mathrm{dyn}} & = & \mathsf{let}\ \langle f,\, \sigma_1\rangle = [\![e_1]\!]\rho_{\mathrm{dyn}}\sigma_{\mathrm{dyn}}\ \mathsf{in} \\
 & & \mathsf{let}\ \langle v,\, \sigma_2\rangle = [\![e_2]\!]\rho_{\mathrm{dyn}}\sigma_1\ \mathsf{in} \\
 & & f\ v\ \sigma_2
\end{array}$$