1 Modularity

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:

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

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:

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

4 Continuation-Passing Style

Consider the statement if $x \leq 0$ then x else $x+1$. We can think of this as $(\lambda y.\,\mathsf{if}\ y\ \mathsf{then}\ x\ \mathsf{else}\ x+1)\,(x \leq 0)$. To evaluate this, we would first evaluate the argument $x \leq 0$ to obtain a Boolean value, then apply the function $\lambda y.\,\mathsf{if}\ y\ \mathsf{then}\ x\ \mathsf{else}\ x+1$ to this value. The function $\lambda y.\,\mathsf{if}\ y\ \mathsf{then}\ x\ \mathsf{else}\ x+1$ is called a *continuation*, because it specifies what is to be done with the result of the current computation in order to continue the computation.

Given an expression e, it is possible to transform the expression into a function that takes a continuation k and applies it to the value of e. The transformation is applied recursively. This is called *continuation-passing style* (CPS). There are a number of advantages to this style:

- The resulting expressions have a much simpler evaluation semantics, since the sequence of reductions to be performed is specified by a series of continuations. The next reduction to be performed is always uniquely determined, and the remainder of the computation is handled by a continuation. Thus evaluation contexts are not necessary to specify the evaluation order.
- In practice, function calls and function returns can be handled in a uniform way. Instead of returning, the called function simply calls the continuation.
- In a recursive function, any computation to be performed on the value returned by a recursive call can be bundled into the continuation. Thus every recursive call becomes tail-recursive. For example, the factorial function

  $$\mathsf{fact}\, n \quad = \quad \mathsf{if}\ n = 0\ \mathsf{then}\ 1\ \mathsf{else}\ n * \mathsf{fact}\,(n-1)$$

  becomes

  $$\mathsf{fact}'\, n\, k \quad = \quad \mathsf{if}\ n = 0\ \mathsf{then}\ k\, 1\ \mathsf{else}\ \mathsf{fact}'\,(n-1)\,(\lambda v.\, k\,(n * v)).$$

  One can show inductively that $\mathsf{fact}'\, n\, k = k\,(\mathsf{fact}\, n)$, therefore $\mathsf{fact}'\, n\, \lambda x.\, x = \mathsf{fact}\, n$. This transformation effectively trades stack space for heap space in the implementation.
- Continuation-passing gives a convenient mechanism for non-local flow of control, such as goto statements and exception handling.

5 CPS Semantics

Our grammar for the λ-calculus was:

$$e \quad ::= \quad x \ \mid\ \lambda x.\, e \ \mid\ e_0\, e_1$$

Our grammar for the CPS λ-calculus will be:

$$v \quad ::= \quad x \ \mid\ \lambda x.\, e \qquad\qquad e \quad ::= \quad v_0\, v_1 \cdots v_n$$

This is a highly constrained syntax. Barring reductions inside the scope of a λ-abstraction operator, the expressions v are all irreducible. The only reducible expression is $v_0\, v_1 \cdots v_n$. If $n \geq 1$, there exactly one redex $v_0\, v_1$, and both the function and the argument are already fully reduced. The small step semantics has a single rule

$$(\lambda x.\, e)\, v \quad \to \quad e\{v/x\},$$

and we do not need any evaluation contexts.

The big step semantics is also quite simple, with only a single rule:

$$\frac{e\{v/x\} \Downarrow v'}{(\lambda x.\, e)\, v \Downarrow v'}.$$

The resulting proof tree will not be very tree-like. The rule has one premise, so a proof will be a stack of inferences, each one corresponding to a step in the small-step semantics. This allows for a much simpler interpreter that can work in a straight line rather than having to make multiple recursive calls.

The fact that we can build a simpler interpreter for the language is a strong hint that this language is lower-level than the λ-calculus. Because it is lower-level (and actually closer to assembly code), CPS is typically used in functional language compilers as an intermediate representation. It also is a good code representation if one is building an interpreter.

6 CPS Conversion

Despite the restricted syntax of CPS, we have not lost any expressive power. Given a λ-calculus expression e, it is possible to define a translation $[\![e]\!]$ that translates it into CPS. This translation is known as *CPS conversion*. It was first described by John Reynolds. The translation takes an arbitrary λ-term e and produces a CPS term $[\![e]\!]$, which is a function that takes a continuation k as an argument. Intuitively, $[\![e]\!]\,k$ applies k to the value of e.

We want our translation to satisfy $e \xrightarrow[\text{CBV}]{*} v$ iff $[\![e]\!]\,k \xrightarrow[\text{CPS}]{*} [\![v]\!]\,k$ for primitive values v and any variable $k \notin FV(e)$, and $e \Uparrow_{\text{CBV}}$ iff $[\![e]\!]\,k \Uparrow_{\text{CPS}}$.

The translation is (adding numbers as primitive values):

$$\begin{array}{rcl} [\![n]\!]\,k & \triangleq & k\,n \\ [\![x]\!]\,k & \triangleq & k\,x \\ [\![\lambda x.\,e]\!]\,k & \triangleq & k\,(\lambda x.\,[\![e]\!]) \quad = \quad k\,(\lambda x k'.\,[\![e]\!]\,k') \\ [\![e_0\,e_1]\!]\,k & \triangleq & [\![e_0]\!](\lambda f.\,[\![e_1]\!](\lambda v.\,f\,v\,k)). \end{array}$$

(Recall $[\![e]\!]\,k \triangleq e'$ really means $[\![e]\!] \triangleq \lambda k.\,e'$.)

6.1 An Example

In the CBV λ-calculus, we have

$$(\lambda x y.\,x)\,1 \quad \to \quad \lambda y.\,1$$

Let's evaluate the CPS-translation of the left-hand side using the CPS evaluation rules.

$$\begin{array}{rcl} [\![(\lambda x y.\,x)\,1]\!]\,k & = & [\![\lambda x.\,\lambda y.\,x]\!](\lambda f.\,[\![1]\!](\lambda v.\,f\,v\,k)) \\ & = & (\lambda f.\,[\![1]\!](\lambda v.\,f\,v\,k))\,(\lambda x.\,[\![\lambda y.\,x]\!]) \\ & \to & [\![1]\!](\lambda v.\,(\lambda x.\,[\![\lambda y.\,x]\!])\,v\,k) \\ & = & (\lambda v.\,(\lambda x.\,[\![\lambda y.\,x]\!])\,v\,k)\,1 \\ & \to & (\lambda x.\,[\![\lambda y.\,x]\!])\,1\,k \\ & \to & [\![\lambda y.\,1]\!]\,k. \end{array}$$

7 CPS and Strong Typing

Now let us use CPS semantics to augment our previously defined FL language translation so that it supports runtime type checking. This time our translated expressions will be functions of ρ and k denoting an environment and a continuation, respectively. The term $\mathcal{E}[\![e]\!]\,\rho\,k$ represents a program that evaluates e in the environment ρ and sends the resulting value to the continuation k.

As before, assume that we have an encoding of variable names x and a representation of environments ρ along with lookup and update functions lookup ρ "x" and update ρ v "x".

In addition, we want to catch type errors that may occur during evaluation. As before, we use integer tags to keep track of types:

$$\mathsf{Err} \triangleq 0 \qquad \mathsf{Null} \triangleq 1 \qquad \mathsf{Bool} \triangleq 2 \qquad \mathsf{Num} \triangleq 3 \qquad \mathsf{Pair} \triangleq 4 \qquad \mathsf{Fun} \triangleq 5$$

A *tagged value* is a value paired with its type tag; for example, $(\mathsf{Bool}, \mathsf{true})$. Using these tagged values, we can now define a translation that incorporates runtime type checking:

$$\begin{aligned}
\mathcal{E}[\![x]\!]\rho k &\triangleq k\,(\mathsf{lookup}\ \rho\ \text{“}x\text{”}) \\
\mathcal{E}[\![b]\!]\rho k &\triangleq k\,(\mathsf{Bool}, b) \\
\mathcal{E}[\![n]\!]\rho k &\triangleq k\,(\mathsf{Num}, n) \\
\mathcal{E}[\![()]\!]\rho k &\triangleq k\,(\mathsf{Null}, \mathsf{nil}) \\
\mathcal{E}[\![(e_1, \ldots, e_n)]\!]\rho k &\triangleq \mathcal{E}[\![e_1]\!]\rho(\lambda x_1.\, \mathcal{E}[\![(e_2, \ldots, e_n)]\!]\rho(\lambda x_2.\, k\,(\mathsf{Pair}, (x_1, x_2)))),\ n \geq 1 \\
\mathcal{E}[\![\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2]\!]\rho k &\triangleq \mathcal{E}[\![e_1]\!]\rho(\lambda p.\, \mathcal{E}[\![e_2]\!](\mathsf{update}\ \rho\ p\ \text{“}x\text{”})\,k) \\
\mathcal{E}[\![\lambda x.\, e]\!]\rho k &\triangleq k\,(\mathsf{Fun}, \lambda v k'.\, \mathcal{E}[\![e]\!](\mathsf{update}\ \rho\ v\ \text{“}x\text{”})\,k') \\
\mathcal{E}[\![\mathsf{error}]\!]\rho k &\triangleq k\,(\mathsf{Err}, 0).
\end{aligned}$$

Now a function application can check that it is actually applying a function:

$$\mathcal{E}[\![e_0\, e_1]\!]\rho k \triangleq \mathcal{E}[\![e_0]\!]\rho(\lambda p.\, \mathsf{let}\ (t, f) = p\ \mathsf{in}\ \mathsf{if}\ t \neq \mathsf{Fun}\ \mathsf{then}\ k\,(\mathsf{Err}, 0)\ \mathsf{else}\ \mathcal{E}[\![e_1]\!]\rho(\lambda v.\, f v k))$$

We can simplify this by defining a helper function check-fn:

$$\mathsf{check\text{-}fn} \triangleq \lambda k p.\, \mathsf{let}\ (t, f) = p\ \mathsf{in}\ \mathsf{if}\ t \neq \mathsf{Fun}\ \mathsf{then}\ k\,(\mathsf{Err}, 0)\ \mathsf{else}\ k f.$$

The helper function takes in a continuation and a tagged value, checks the type, strips off the tag, and passes the raw (untagged) value to the continuation. Then

$$\mathcal{E}[\![e_0\, e_1]\!]\rho k \triangleq \mathcal{E}[\![e_0]\!]\rho(\mathsf{check\text{-}fn}\,(\lambda f.\, \mathcal{E}[\![e_1]\!]\rho(\lambda v.\, f v k))).$$

Similarly,

$$\begin{aligned}
\mathcal{E}[\![\mathsf{if}\ e_0\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2]\!]\rho k &\triangleq \mathcal{E}[\![e_0]\!]\rho(\mathsf{check\text{-}bool}\,(\lambda b.\, \mathsf{if}\ b\ \mathsf{then}\ \mathcal{E}[\![e_1]\!]\rho k\ \mathsf{else}\ \mathcal{E}[\![e_2]\!]\rho k)) \\
\mathcal{E}[\![\#n\, e]\!]\rho k &\triangleq \mathcal{E}[\![e]\!]\rho(\mathsf{check\text{-}pair}\,(\lambda t.\, k\,(\#n\, t))).
\end{aligned}$$

8 CPS Semantics for FL!

8.1 Syntax

Since FL! has references, we need to add a store σ to our notation. Thus we now have translations with the form $\mathcal{E}[\![e]\!]\rho k\sigma$, which means, "Evaluate e in the environment ρ with store σ and send the resulting value and the new store to the continuation k." A continuation is now a function of a value and a store; that is, a continuation k should have the form $\lambda v\sigma.\ \cdots$.

The translation is:

- Variable: $\mathcal{E}[\![x]\!]\rho k\sigma \triangleq k\,(\mathsf{lookup}\ \rho\ \text{“}x\text{”})\,\sigma$.
  If we think about this translation as a function and η-reduce away the σ, we obtain

$$\mathcal{E}[\![x]\!]\rho k \;=\; \lambda\sigma.\, k\,(\mathsf{lookup}\ \rho\ \text{“}x\text{”})\,\sigma \;=\; k\,(\mathsf{lookup}\ \rho\ \text{“}x\text{”}).$$

Note that in the η-reduced version, we have the same translation that we for FL. In general, any expression in FL! that is not state-aware can be η-reduced to the same translation as FL. Thus in order to translate to FL!, we need to add translation rules only for the functionality that is state-aware.

We assume that we have a type tag Loc for locations and check-loc for tagging values as locations and checking those tags. We also assume that we have extended our lookup and update functions to apply to stores.

$$\begin{aligned}
\mathcal{E}[\![\mathsf{ref}\, e]\!]\,\rho\, k\,\sigma &\triangleq \mathcal{E}[\![e]\!]\,\rho\,(\lambda v\sigma'.\,\mathsf{let}\ (\ell,\sigma'') = (\mathsf{malloc}\,\sigma'\,v)\ \mathsf{in}\ k\,(\mathsf{Loc},\,\ell)\,\sigma'')\,\sigma \\
\mathcal{E}[\![!e]\!]\,\rho\, k &\triangleq \mathcal{E}[\![e]\!]\,\rho\,(\mathsf{check\text{-}loc}\,(\lambda\ell\sigma'.\,k\,(\mathsf{lookup}\ \sigma'\ \text{“}\ell\text{”})\,\sigma')) \\
\mathcal{E}[\![e_1 := e_2]\!]\,\rho\, k &\triangleq \mathcal{E}[\![e_1]\!]\,\rho\,(\mathsf{check\text{-}loc}\,(\lambda\ell.\,\mathcal{E}[\![e_2]\!]\,\rho\,(\lambda v\sigma'.\,k\,(\mathsf{Null},\mathsf{nil})\,(\mathsf{update}\ \sigma'\ v\ \text{“}\ell\text{”}))))).
\end{aligned}$$

References

[1] Glynn Winskel. *The Formal Semantics of Programming Languages.* MIT Press, 1993.