1 Non-hierarchical Scope: Modules

The binding constructs we have seen so far are all hierarchical in nature. Each construct establishes a parent-child relationship between an outer context in which the declaration is not visible and an inner (body) context in which the declaration is visible. In static scoping, the hierarchy is determined by the abstract syntax tree, while in dynamic scoping, the hierarchy is determined by the tree of procedure calls generated at run-time. In both these scoping mechanisms, there is no natural way to communicate a declaration laterally across the tree-structure imposed by the hierarchy.

For small programs, this is not ordinarily a problem, but when a large program is broken into independent pieces, or modules, the constraint of hierarchy can be a problem. Modules connect and communicate with each other via collections of bindings; a module provides services by exporting a set of bindings, and makes use of other modules' services by importing bindings from those other modules. In a hierarchical language, the scope of a binding is a single region of program, so all the clients of a module must reside in the region where the module's bindings are in scope.

The traditional solution to the problem of communicating modules is to use a global namespace. All exported bindings from all modules are defined in a single environment, so all exported bindings are available to all modules. This approach is used in languages like C or FORTRAN. A C program is a bunch of top-level functions and in this way it is possible to call any function anywhere. A global namespace has some major drawbacks: In order to avoid accidental name collisions, every module must be aware of all definitions made by all other modules, even those definitions that are completely irrelevant; the dependencies between modules are poorly documented, making intermodule dependencies difficult to track, and leading to fragile code over time.

A way for languages to overcome the hierarchical scoping of binding constructs is to provide a value with named subparts. For this purpose, we will introduce a new module value that bundles up a set of bindings at one point in a program and can communicate them to a point that is related neither lexically nor dynamically to the declarations of those bindings, Typically, a module defines a set of named values, especially procedures, that provide a particular function. Some examples of these are: modules in ML, and classes in C++ and Java.

To introduce a simple module mechanism into uF, we have to add new expression forms to uF. Expressions are then defined by the following grammar:

$$e \quad ::= \quad ... \quad | \quad \mathsf{module}\ (x_1 = e_1, ..., x_n = e_n) \quad | \quad e_m.x \quad | \quad \mathsf{with}\ e_m\ e$$

We need to extend the domain equations to allow new kinds of values – module values.

$$\begin{aligned} \mathit{Value} &= ... + \mathit{Module} \\ \mathit{Module} &= \mathit{Env} \\ \mathit{Env} &= \mathit{Var} \to \mathit{Value} + \mathit{Error} \end{aligned}$$

The denotational semantics of modular uF now needs to include the semantics for the module constructs:

$$\begin{aligned} \mathcal{C}[\![\mathsf{module}\ (x_1 = e_1, ..., x_n = e_n)]\!] \quad = \quad & \lambda\rho\ .\ \mathsf{let}\ v_1 = \mathcal{C}[\![e_1]\!]\rho\ ... \\ & \qquad \mathsf{let}\ v_n = \mathcal{C}[\![e_n]\!]\rho\ . \\ & \qquad\quad \rho_E[x_1 \mapsto v_1,\ ...\ , x_n \mapsto v_n] \end{aligned}$$

$$
\begin{array}{rcl}
 & & (\rho_E = \lambda x \ .\ error) \\
\mathcal{C}[\![e_m.x]\!]\rho & = & \mathsf{let}\ m = \mathcal{C}[\![e_m]\!]\rho \ .\ m(x) \\
\mathcal{C}[\![\mathsf{with}\ e_m\ e]\!]\rho & = & \mathsf{let}\ m = \mathcal{C}[\![e_m]\!]\rho \ .\ \mathcal{C}[\![e]\!](\lambda x \in Var \ . \\
 & & \quad \mathsf{case}\ m(x)\ \mathsf{of}\ Value(v)\ .\ v \\
 & & \quad\quad \mid Error(e)\ .\ \rho(x))
\end{array}
$$

2 Converting uF to a lazy language

So far, we have been talking about languages like Scheme and ML with strict evaluation. Now we want to define a lazy uF language like Haskell and Miranda. We are going to keep the syntax of uF exactly the same as before, and reinterprete the expressions so that they become lazy. We do not evaluate anything unless we absolutely have to. First, we need to write down the domain equations to represent values in the language.

$$
\begin{array}{rcl}
Value & = & \mathcal{Z} + \mathcal{T} + \mathcal{U} + Pair + Function \\
Result & = & (Value + Error)_\perp \\
Pair & = & Result \times Result \\
Binding & = & Result \\
Function & = & Binding \to Result \\
Env & = & Variable \to Result
\end{array}
$$

Just as in strict uF, the semantic function $\mathcal{C}$ takes an expression and maps it to a function that produces a result given an environment in which to interprete the free viariables of the expression:

$$\mathcal{C}[\![e]\!] \in Env \to Result$$

Since the language is lazy, we can say something like:

$$
\begin{array}{rcl}
\mathsf{let\ loop} & = & (\ \mathsf{fn}\ x\ (\ x\ x\ )\ )\ (\ \mathsf{fn}\ x\ (\ x\ x\ )\ ) \\
 & & \mathsf{in\ let}\ p\ =\ \langle\ \mathsf{loop}\ ,\ \mathsf{loop}\ \rangle \\
 & & \quad \mathsf{in}\ 0
\end{array}
$$

This expression will evaluate to zero because the language is lazy and will not evaluate the loop or the elements of the pair p. In the original strict uF, the program will fail to terminate while evaluating loop.

The denotational semantics of the lazy uF language is:

$$
\begin{array}{rcl}
\mathcal{C}[\![x]\!]\rho & = & \rho\ (\ x\ ) \\
\mathcal{C}[\![\langle\ e_1\ ,\ e_2\ \rangle]\!]\rho & = & \langle\ \mathcal{C}[\![e_1]\!]\rho\ ,\ \mathcal{C}[\![e_2]\!]\rho\ \rangle \\
\mathcal{C}[\![\mathsf{fn}\ x\ e]\!]\rho & = & \lambda r \in Result\ .\ \mathcal{C}[\![e]\!]\rho[x \mapsto r] \\
\mathcal{C}[\![e_1\ e_2]\!]\rho & = & \mathsf{scase}\ \mathcal{C}[\![e_1]\!]\rho\ \mathsf{of} \\
 & & \quad Function(f)\ .\ f\ (\ \mathcal{C}[\![e_2]\!]\rho\ ) \\
\mathcal{C}[\![\mathsf{rec}\ x\ e_r]\!]\rho & = & fix\ \lambda r \in Result\ .\ \mathcal{C}[\![e_r]\!]\rho\ [x \mapsto r]
\end{array}
$$

The rules for first, rest and arithmetic operations will not change and are still strict evaluations. In this language, we can have $e_r = e$. The fix operator can be applied to not just functions, but also other things like tuples. We can say something like ones:

$$ones \equiv \mathsf{rec}\ x\ \langle\ 1\ ,\ x\ \rangle$$

The value *ones* represents an infinite sequence of ones:

$$\langle\ 1\ ,\ \langle\ 1\ ,\ \langle\ 1\ ,\ ...\ \rangle\ \rangle\ \rangle$$

The denotational semantics of ones is obtained by applying the *fix* operator to function F:

$$\mathcal{F} = \lambda r \in Result\ .\ \langle\ 1\ ,\ r\ \rangle$$

$$\mathcal{C}[\![\mathsf{rec}\ x\ \langle\ 1\ ,\ x\ \rangle]\!]\rho = fix\ \lambda r \in Result\ .\ \langle\ 1\ ,\ r\ \rangle$$

$$\begin{aligned}
\mathcal{F}^0\ (\ \bot\ ) &= \bot \\
\mathcal{F}^1\ (\ \bot\ ) &= \langle\ 1\ ,\ \bot\ \rangle \\
\mathcal{F}^2\ (\ \bot\ ) &= \langle\ 1\ ,\ \langle\ 1\ ,\ \bot\ \rangle\ \rangle \\
&\ \ \vdots
\end{aligned}$$

The use of *fix* requires that $r \in Result$ is a pointed CPO, and we require that the function $\mathcal{C}[\![e_r]\!]\rho[x \mapsto r]$ to be continuous. In order to make fixed-point in *ones* construction, *Result* must be a CPO. This requires that Pair is also a CPO. Therefore, Pair must allow infinite sequence construction, unlike the least fixed-point solutions we saw how to derive earlier for inductive definitions.

3 lazy uF to strict uF definitional Semantics

Our call-by-name lazy semantics are easier to write down than the strict semantics. This seems to tell us that a lazy language is easier to compile than a strict language, which is not the case. How do we find out what is involved in compiling a lazy language? We can write a definitional semantics for our lazy uF language in terms of the strict language.

$$\mathcal{D}[\![e]\!] = e'$$

where e is written in lazy uF, and e' is written in strict uF.In 1961, it was figured out for the language Algol 60 that we can implement laziness using thunks. In order to write a value, we turn it into a function, which if we call it, will give us the value that you want. If we want to evaluate a value e, we apply a dummy argument to the function:

$$\mathcal{D}[\![e]\!]\ \#\mathsf{u}$$

This is the "strict e". The "lazy e" will be a thunk:

$$\mathcal{D}[\![e]\!]$$

Now we can write down the definitional semantics:

$$\begin{aligned}
\mathcal{D}[\![n]\!] &= (\ \mathsf{fn}\ u\ n\ ) \\
\mathcal{D}[\![x]\!] &= x \\
\mathcal{D}[\![\langle\ e_1\ ,\ e_2\ \rangle]\!] &= (\ \mathsf{fn}\ u\ \langle\ \mathcal{D}[\![e_1]\!]\ ,\ \mathcal{D}[\![e_2]\!]\ \rangle\ )
\end{aligned}$$

$$
\begin{array}{rcl}
\mathcal{D}[\![\mathsf{first}\ e]\!] & = & \mathsf{first}\ (\ \mathcal{D}[\![e]\!]\ \#\mathsf{u}\ ) \\
\mathcal{D}[\![\mathsf{fn}\ x\ e]\!] & = & (\ \mathsf{fn}\ u\ (\ \mathsf{fn}\ x\ \mathcal{D}[\![e]\!]\ )\ ) \\
\mathcal{D}[\![e_1\ e_2]\!] & = & (\ \mathcal{D}[\![e_1]\!]\ \#\mathsf{u}\ )\ \mathcal{D}[\![e2]\!] \\
\mathcal{D}[\![\mathsf{rec}\ x\ e]\!] & = & \mathsf{rec}\ x\ \mathcal{D}[\![e]\!]
\end{array}
$$

To see what's going on here, we can apply this translation to rec x ⟨1,x⟩:

$$
\begin{array}{rcl}
\mathcal{D}[\![\mathsf{rec}\ x\ \langle\ 1\ ,\ x\ \rangle]\!] & = & \mathsf{rec}\ x\ (\ \mathcal{D}[\![\langle\ 1\ ,\ x\ \rangle]\!]\ ) \\
 & & \mathsf{rec}\ x\ (\ \mathsf{fn}\ u\ \langle\ \mathcal{D}[\![1]\!]\ ,\ \mathcal{D}[\![x]\!]\ \rangle\ ) \\
 & & \mathsf{rec}\ x\ (\ \mathsf{fn}\ u\ \langle\ (\ \mathsf{fn}\ u\ 1\ )\ ,\ x\ \rangle\ ) \\
 & & \\
\mathcal{D}[\![\mathsf{first}\ (\mathsf{rec}\ x\ \langle\ 1\ ,\ x\ \rangle)]\!] & = & \mathsf{first}\ (\mathsf{rec}\ x\ (\ \mathsf{fn}\ u\ \langle\ (\ \mathsf{fn}\ u\ 1\ )\ ,\ x\ \rangle\ )\ \#\mathsf{u}) \\
 & \to & \mathsf{first}\ ((\ \mathsf{fn}\ u\ \langle\ \mathsf{fn}\ u\ 1\ ,\ \mathsf{rec}\ x\ (\ \mathsf{fn}\ u\ \langle\ \mathsf{fn}\ u\ 1\ ,\ x\ \rangle)\rangle)\ \#\mathsf{u}) \\
 & \to & \mathsf{first}\ (\langle\ \mathsf{fn}\ u\ 1\ ,\ \mathsf{rec}\ x\ (\ \mathsf{fn}\ u\ \langle\ \mathsf{fn}\ u\ 1\ ,\ x\ \rangle)\rangle) \\
 & \to & \mathsf{first}\ (\langle\ \mathsf{fn}\ u\ 1\ ,\ \mathsf{fn}\ u\ \langle\ \mathsf{fn}\ u\ 1\ ,\ \mathsf{rec}\ x\ (\ \mathsf{fn}\ u\ \langle\ \mathsf{fn}\ u\ 1\ ,\ x\ \rangle)\rangle\rangle) \\
 & \to & \mathsf{fn}\ u\ 1 \\
 & = & \mathcal{D}[\![1]\!] \\
 & & \\
\mathcal{D}[\![\mathsf{rest}\ (\mathsf{rec}\ x\ \langle\ 1\ ,\ x\ \rangle)]\!] & = & \mathsf{rest}\ (\mathsf{rec}\ x\ (\ \mathsf{fn}\ u\ \langle\ (\ \mathsf{fn}\ u\ 1\ )\ ,\ x\ \rangle\ )\ \#\mathsf{u}) \\
 & \to & \mathsf{rest}\ ((\ \mathsf{fn}\ u\ \langle\ \mathsf{fn}\ u\ 1\ ,\ \mathsf{rec}\ x\ (\ \mathsf{fn}\ u\ \langle\ \mathsf{fn}\ u\ 1\ ,\ x\ \rangle)\rangle)\ \#\mathsf{u}) \\
 & \to & \mathsf{rest}\ (\langle\ \mathsf{fn}\ u\ 1\ ,\ \mathsf{rec}\ x\ (\ \mathsf{fn}\ u\ \langle\ \mathsf{fn}\ u\ 1\ ,\ x\ \rangle)\rangle) \\
 & \to & \mathsf{rest}\ (\langle\ \mathsf{fn}\ u\ 1\ ,\ \mathsf{fn}\ u\ \langle\ \mathsf{fn}\ u\ 1\ ,\ \mathsf{rec}\ x\ (\ \mathsf{fn}\ u\ \langle\ \mathsf{fn}\ u\ 1\ ,\ x\ \rangle)\rangle\rangle) \\
 & \to & \mathsf{fn}\ u\ \langle\ \mathsf{fn}\ u\ 1\ ,\ \mathsf{rec}\ x\ (\ \mathsf{fn}\ u\ \langle\ \mathsf{fn}\ u\ 1\ ,\ x\ \rangle)\rangle \\
 & = & \mathcal{D}[\![\mathsf{rec}\ x\ \langle\ 1\ ,\ x\ \rangle]\!]
\end{array}
$$