1 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.

1.1 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.

2 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!

3 The FL! Language

3.1 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:

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

3.2 The Store

We define a *store* as a partial function $\sigma : Loc \rightharpoonup Val$ 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:

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

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 $\mathsf{dom}\,\sigma$.

3.3 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!.

4 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 $\mathsf{let}\ \langle x,\, \sigma'\rangle = [\![e_0]\!]\rho\sigma\ \mathsf{in}\ \ldots$ is syntactic sugar for

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

$$
\begin{array}{rcl}
[\![n]\!] \rho\sigma & = & \langle n,\, \sigma \rangle \\
[\![x]\!] \rho\sigma & = & \langle \rho(x),\, \sigma \rangle \\
[\![\textsf{if } e_0 \textsf{ then } e_1 \textsf{ else } e_2]\!] \rho\sigma & = & \textsf{let } \langle x_0,\, \sigma' \rangle = [\![e_0]\!] \rho\sigma \textsf{ in} \\
 & & \textsf{if } x_0 \textsf{ then } [\![e_1]\!] \rho\sigma' \textsf{ else } [\![e_2]\!] \rho\sigma' \\
[\![\textsf{ref } e]\!] \rho\sigma & = & \textsf{let } \langle x',\, \sigma' \rangle = [\![e]\!] \rho\sigma \textsf{ in } \textsf{malloc}\, \sigma'\, x' \\
[\![!e]\!] \rho\sigma & = & \textsf{let } \langle x',\, \sigma' \rangle = [\![e]\!] \rho\sigma \textsf{ in } \langle \textsf{lookup}\, \sigma'\, x',\, \sigma' \rangle \\
[\![e_1 := e_2]\!] \rho\sigma & = & \textsf{let } \langle x_1,\, \sigma_1 \rangle = [\![e_1]\!] \rho\sigma \textsf{ in} \\
 & & \textsf{let } \langle x_2,\, \sigma_2 \rangle = [\![e_2]\!] \rho\sigma_1 \textsf{ in} \\
 & & \langle \textsf{null},\, \mathrm{update}\ \sigma_2\ x_1\ x_2 \rangle \\
[\![e_1 \,;\, e_2]\!] \rho\sigma & = & \textsf{let } \langle x,\, \sigma_1 \rangle = [\![e_1]\!] \rho\sigma \textsf{ in } [\![e_2]\!] \rho\sigma_1 \qquad x \notin FV(e_2) \\
[\![\lambda x.\, e]\!] \rho_{\mathrm{lex}} \sigma_{\mathrm{lex}} & = & \langle \lambda v \sigma_{\mathrm{dyn}} .\, [\![e]\!] \rho_{\mathrm{lex}}[v/x]\, \sigma_{\mathrm{dyn}},\, \sigma_{\mathrm{lex}} \rangle \\
[\![e_1\ e_2]\!] \rho_{\mathrm{dyn}} \sigma_{\mathrm{dyn}} & = & \textsf{let } \langle x_1,\, \sigma_1 \rangle = [\![e_1]\!] \rho_{\mathrm{dyn}} \sigma_{\mathrm{dyn}} \textsf{ in} \\
 & & \textsf{let } \langle x_2,\, \sigma_2 \rangle = [\![e_2]\!] \rho_{\mathrm{dyn}} \sigma_1 \textsf{ in} \\
 & & x_1\ x_2\ \sigma_2
\end{array}
$$