Program state refers to the ability to change program variables over time. We saw this feature in Imp (which was conspicuously missing functions) but then it went away in λ-calculus (which was conspicuously missing everything but functions). Even Fl, which reintroduced most of the programming constructs, still did not have state. Once a variable was bound, there was no way to change its value as long as it was in scope.

State is not a necessary feature in a programming language—after all, λ-calculus is Turing complete despite no notion of state—but it is common in most languages and many programmers are accustomed to it.

Programming Paradigms. Two major programming paradigms are *functional* (stateless) and *imperative* (stateful). In a purely functional language, expressions resemble mathematical formulae. This structure allows programmers to reason equationally and not worry about where else a function is used or how many times something may be run. As a result, it avoids many of the pitfalls associated with a constantly changing execution environment.

Concurrency, in particular, is much simpler with functional programming. Confluence (the Church–Rosser property) means it does not matter which order operations execute, at the end the result will be the same. There can be no race conditions or inconsistent views of the world.

On the other hand, imperative programming more closely resembles the way we perceive the world and, especially, the operations happening inside of a real computer. There exists some underlying notion of state (of the world, of memory, of storage, etc), and that state can change over time.

1 Mutable References

Instead of allowing variable assignment directly, we will work with *mutable references* (aka *pointers*). These references 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.

$$\begin{aligned}&\mathsf{let}\ x = \mathsf{ref}\ 1\ \mathsf{in}\\&\mathsf{let}\ y = x\ \mathsf{in}\\&\quad x := 2\ ;\ !y\end{aligned}$$

In this code, ref 1 allocates a new reference pointing to the value 1 and returns the reference. So the first line creates this reference and assigns x to the reference. The second line assigns y to x, meaning both variables are bound to the same reference. Then we update the value pointed to by x to 2, and finally dereference y with $!y$. Because x and y point to the same place, modifying the value one points to also modifies the value the other points to, so this program will return 2. When you kick x, y jumps!

Reference should not be confused with mutable variables. A variable is *mutable* if its binding can change. The difference is subtle: variables are bound to values in an environment, and if the variable is mutable, it can be rebound to a different value. With references, the variable itself is bound to a *location*. The location is mutable—it can be rebound to a different value—but the variable itself is not. In Imp and imperative languages such as C, variables are typically mutable, whereas in functional languages such as Fl, OCaml, and Haskell, they are typically not.

2 The FL! Language

To see how mutable references work in an otherwise-functional language, we will add them to the FL language we defined previously to create a new language FL!.[1] All FL expressions are also expressions of FL!, and there are a few more. Note that there is a set **Loc** of *memory locations* that we denote ℓ. The syntax is as follows.

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

The "$\cdots$" is used to indicate that we are extending the BNF grammar from FL without having to write out everything again.

2.1 Small-Step Operational Semantics

Unlike in λ-calculus or functional FL, we can no longer define our semantics entirely by substitution.

Instead we will use a store to keep track of the mappings of memory locations for our references. This store is extremely similar to stores we saw previously in the semantics for IMP. It is a partial function $\sigma : \mathbf{Loc} \rightharpoonup \mathbf{Val}$ from memory locations to FL! values, and we require that it have a finite domain (that is, only finitely many location have mappings). We will again use the standard rebinding operator $\sigma[\ell \mapsto v]$. We now define the small-step operational semantics on *configurations*, $\langle e, \sigma \rangle$, just as we did for IMP.

We will again simplify our evaluation order rules by using evaluation contexts. Just as we extended the FL grammar for expressions, we extend the FL grammar for evaluation contexts.

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

Note that the hole $[\cdot]$ is already included with the $\cdots$, so we do not need to write it again.

The operational semantic rule for using these contexts must change slightly. Before, our small step relation was over FL expressions, but now it is over entire configurations. Because of the nature of state, if the state changes when stepping an expression e, we need to retain that change when stepping e inside a larger context. The rule is therefore

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

We also need to add reduction rules for these new forms. Those rules are as follows.

$$[\textsc{New}]\ \frac{\ell \notin \mathrm{dom}(\sigma)}{\langle \mathsf{ref}\ v, \sigma \rangle \longrightarrow \langle \ell, \sigma[\ell \mapsto v] \rangle} \qquad [\textsc{Assign}]\ \frac{\ell \in \mathrm{dom}(\sigma)}{\langle \ell := v, \sigma \rangle \longrightarrow \langle (), \sigma[\ell \mapsto v] \rangle}$$

$$[\textsc{Deref}]\ \frac{\ell \in \mathrm{dom}(\sigma)}{\langle !\ell, \sigma \rangle \longrightarrow \langle \sigma(\ell), \sigma \rangle} \qquad [\textsc{Seq}]\ \frac{}{\langle v\ ;\ e, \sigma \rangle \longrightarrow \langle e, \sigma \rangle}$$

It can be shown that it is impossible to create dangling pointers in FL!. That is, if a program doesn't start with any pointers at all, then every pointer that appears will always be mapped in the store.

2.2 FL! in Action

To see how this operational semantics works, we look at the example from Section 1 and see how it executes.

$$\begin{aligned}\langle \mathsf{let}\ x = \mathsf{ref}\ 1\ \mathsf{in}\ \mathsf{let}\ y = x\ \mathsf{in}\ x := 2\ ;\ !y, \varnothing \rangle &\longrightarrow \langle \mathsf{let}\ x = \ell\ \mathsf{in}\ \mathsf{let}\ y = x\ \mathsf{in}\ x := 2\ ;\ !y, [\ell \mapsto 1] \rangle \\ &\longrightarrow \langle \mathsf{let}\ y = \ell\ \mathsf{in}\ \ell := 2\ ;\ !y, [\ell \mapsto 1] \rangle \\ &\longrightarrow \langle \ell := 2\ ;\ !\ell, [\ell \mapsto 1] \rangle\end{aligned}$$

[1] FL! is pronounced "FL bang." The exclamation point, or *bang*, is a common symbol used to indicate a stateful operation in a primarily functional language.

We can now see that x and y have been substituted for the same location ℓ, and anything that happens to one of them will therefore be reflected in the other. In particular, this configuration continues to step as follows.

$$\langle \ell := 2 \; ; \; !\ell, [\ell \mapsto 1] \rangle \longrightarrow \langle () \; ; \; !\ell, [\ell \mapsto 2] \rangle \longrightarrow \langle !\ell, [\ell \mapsto 2] \rangle \longrightarrow \langle 2, [\ell \mapsto 2] \rangle$$

Exactly as expected, dereferncing y produced the value we assigned into x one operation earlier.

This behavior becomes particularly interesting when a function closes over a reference. For instance, consider the following code that produces a function that will ignore its argument and return how many times it has been called (including this one). The context of those calls is irrelevant, just the number of them.

$$\mathsf{let}\ cnt = \mathsf{ref}\ 0\ \mathsf{in}\ (\lambda\_.\ (cnt := !cnt + 1)\ ;\ !cnt)$$

We could even do something weirder: we could build a pair of functions that each return how many times the *other* has been called.

$$\begin{array}{l} \mathsf{let}\ cnt_1 = \mathsf{ref}\ 0\ \mathsf{in} \\ \mathsf{let}\ cnt_2 = \mathsf{ref}\ 0 \\ \mathsf{in}\ \big((\lambda\_.\ (cnt_1 := !cnt_1 + 1)\ ;\ !cnt_2), \\ \qquad (\lambda\_.\ (cnt_2 := !cnt_2 + 1)\ ;\ !cnt_1)\big) \end{array}$$

Mutable references—and stateful computation in general—allows for some very odd behavior.

3 Translating FL! to FL

Even though we had to modify the structure of the small step operational semantic relation, this addition did not fundamentally add any computational power to the language. To show this, we will build an adequate translation from FL! to FL (though we will not prove its adequacy here).

To track the mutable state within the FL code, we construct an explicit representation of the heap that we call an *environment*, with the same type as a heap: a partial function with a finite domain from locations to values. We need a way to represent this environment in the target language of our transaltion, here FL. To do this, we will encode locations in the environment into values of the target language. We will write $\ulcorner \ell \urcorner$ to denote the encoding of location ℓ into FL. The exact details of the encoding don't really matter as long as we can look up the value of a location given its encoding and update an environment with a new or modified binding. For instance, any structure that allows us to encode differet locations differently and check equality of encoded variable names would work. Numbers are generally a good choice, but they are not the only one.

We do three things with heaps: allocate new locations, update existing locations, and loop up values, so we demand that if S is a representation of heap σ, the following three operations must exist.

1. $\mathsf{alloc}\ S\ v\ =\ (\ulcorner \ell \urcorner, S')$ where $\ell \notin \mathrm{dom}(\sigma)$ and S' is a representation of $\sigma[\ell \mapsto v]$).
2. $\mathsf{update}\ S\ \ulcorner \ell \urcorner\ v$ is a representation of $\sigma[\ell \mapsto v]$
3. $\mathsf{lookup}\ S\ \ulcorner \ell \urcorner\ =\ \begin{cases} \sigma(\ell) & \text{if } \ell \in \mathrm{dom}(\sigma) \\ \mathsf{error} & \text{if } \ell \notin \mathrm{dom}(\sigma) \end{cases}$

To simplify notation, despite a slight risk of confusion, we will use σ to represent both the heap and the representation of that heap in FL, so we will write $\mathsf{alloc}\ \sigma\ v$, $\mathsf{update}\ \sigma\ \ulcorner \ell \urcorner\ v$, and $\mathsf{lookup}\ \sigma\ \ulcorner \ell \urcorner$.

The following translation maps FL! expressions to FL expression. Note that $[\![e]\!]$ represents a function that takes a store σ and produces an FL pair (v, σ') where v is an FL value and σ' is a store. These represent the output value (translated) and final store (representation) obtained by evaluating e. To simplify notation, we

will use the expression $\mathsf{let}\ (x, \sigma') = [\![e]\!]\ \sigma\ \mathsf{in}\ e'$ as syntactic sugar for

$$\begin{array}{l}\mathsf{let}\ p = [\![e]\!]\ \sigma\ \mathsf{in} \\ \mathsf{let}\ x = \#1\ p\ \mathsf{in} \\ \mathsf{let}\ \sigma' = \#2\ p \\ \ \mathsf{in}\ e'\end{array}$$

where $p \notin \mathrm{FV}(e')$. Here is the translation for simple expressions. For simplicity of exposition, we skip tuples, and multi-argument functions, though those cases are not conceptually complicated.

$$\begin{array}{rcl}
[\![x]\!]\ \sigma & \triangleq & (x, \sigma) \\
[\![n]\!]\ \sigma & \triangleq & (n, \sigma) \\
[\![\mathsf{true}]\!]\ \sigma & \triangleq & (\mathsf{true}, \sigma) \\
[\![\mathsf{false}]\!]\ \sigma & \triangleq & (\mathsf{false}, \sigma) \\
[\![\mathsf{if}\ e_0\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2]\!]\ \sigma & \triangleq & \mathsf{let}\ (b, \sigma') = [\![e_0]\!]\ \sigma\ \mathsf{in} \\
 & & \quad \mathsf{if}\ b\ \mathsf{then}\ [\![e_1]\!]\ \sigma'\ \mathsf{else}\ [\![e_2]\!]\ \sigma' \\
[\![\lambda x.\, e]\!]\ \sigma & \triangleq & (\lambda x \tau.\, [\![e]\!]\ \tau, \sigma) \\
[\![\mathsf{rec}\ f(x) = e]\!]\ \sigma & \triangleq & (\mathsf{rec}\ f(x) = \lambda \tau.\, [\![e]\!]\ \tau, \sigma) \\
[\![e_1\ e_2]\!]\ \sigma & \triangleq & \mathsf{let}\ (f, \sigma_1) = [\![e_1]\!]\ \sigma\ \mathsf{in} \\
 & & \quad \mathsf{let}\ (x, \sigma_2) = [\![e_2]\!]\ \sigma_1\ \mathsf{in} \\
 & & \quad f\ x\ \sigma_2 \\
[\![\mathsf{ref}\ e]\!]\ \sigma & \triangleq & \mathsf{let}\ (x, \sigma') = [\![e]\!]\ \sigma\ \mathsf{in}\ (\mathsf{alloc}\ \sigma'\ x) \\
[\![e_1 := e_2]\!]\ \sigma & \triangleq & \mathsf{let}\ (l, \sigma_1) = [\![e_1]\!]\ \sigma\ \mathsf{in} \\
 & & \quad \mathsf{let}\ (x, \sigma_2) = [\![e_2]\!]\ \sigma_1\ \mathsf{in} \\
 & & \quad ((), \mathsf{update}\ \sigma_2\ l\ x) \\
[\![!e]\!]\ \sigma & \triangleq & \mathsf{let}\ (l, \sigma') = [\![e]\!]\ \sigma\ \mathsf{in}\ ((\mathsf{lookup}\ \sigma'\ l), \sigma') \\
[\![e_1\ ;\ e_2]\!]\ \sigma & \triangleq & \mathsf{let}\ (_, \sigma') = [\![e_1]\!]\ \sigma\ \mathsf{in}\ [\![e_2]\!]\ \sigma'
\end{array}$$

Note that the translations of $\lambda x.\, e$ and $\mathsf{rec}\ f(x) = e$ now take an extra argument for the new store that is current at the time of the call. That is because stores do not follow lexical scoping like variables, instead they follow a *dynamic scope* discipline whereby a location cares about the value of the store when it is used, not when it is defined. That is also why we need to track the store explicitly in the operational semantics, rather than doing using substitutions like we do for lexically-scoped variables.