1 Introduction

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.

2 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

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

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.

3 References

References (aka *pointers*) provide another level of mutable state. 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:

```
let x = ref 1 in
let y = x in
x := 2; !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 for the same data value. 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 immutable. In IMP and imperative languages such as C, variables are typically mutable, whereas in functional languages such as FL and OCaml, they are typically not.

4 The FL! Language

4.1 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:

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

4.2 The Store

We define a *store* as a partial function $\sigma : \mathit{Loc} \rightharpoonup \mathit{Val}$ 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. As with environments, we have a *rebinding operator* on stores:

$$\mathsf{dom}\,\sigma[v/\ell] = \mathsf{dom}\,\sigma \cup \{\ell\} \qquad\qquad \sigma[v/\ell](\ell') = \begin{cases} v & \text{if } \ell = \ell', \\ \sigma(\ell') & \text{if } \ell \neq \ell' \text{ and } \ell' \in \mathsf{dom}\,\sigma, \\ \text{undefined} & \text{if } \ell \neq \ell' \text{ and } \ell' \notin \mathsf{dom}\,\sigma. \end{cases}$$

Thus $\sigma[v/\ell]$ refers to σ with the location ℓ rebound to the value v if $\ell \in \mathsf{dom}\,\sigma$, otherwise it refers to σ with the new location ℓ bound to v and added to $\mathsf{dom}\,\sigma$.

4.3 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:

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

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

$$\langle \mathsf{ref}\ v,\, \sigma\rangle \to \langle \ell,\, \sigma[v/\ell]\rangle,\ \ \ell \notin \mathsf{dom}\,\sigma \qquad\qquad \langle !\,\ell,\, \sigma\rangle \to \langle \sigma(\ell),\, \sigma\rangle,\ \ \ell \in \mathsf{dom}\,\sigma$$

$$\langle \ell := v,\, \sigma\rangle \to \langle \mathsf{null},\, \sigma[v/\ell]\rangle,\ \ \ell \in \mathsf{dom}\,\sigma \qquad\qquad \langle v \,;\, e,\, \sigma\rangle \to \langle e,\, \sigma\rangle.$$

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

5 Translating FL! to FL

We can give an adequate translation of FL! to FL. Using the same mechanisms available in FL that we used in Lecture 11 to implement environments, we can implement stores and the following operations on them:

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

Here $\ulcorner \ell \urcorner$ denotes the representation in FL of a location ℓ. At the risk of confusion, as we did with environments, we will denote by σ both the store σ in $\mathsf{FL!}$ and its representation in FL.

The following translation maps an $\mathsf{FL!}$ expression e to an FL expression $[\![e]\!]$. The expression $[\![e]\!]$ represents a function that takes an environment ρ and store σ and produces an FL pair (v, σ'), where v is an FL value and σ' is a store. The expression $\mathsf{let}\ (v, \sigma') = [\![e]\!]\,\rho\sigma\ \mathsf{in}\ \ldots$ is syntactic sugar for

$$\mathsf{let}\ x = [\![e]\!]\,\rho\sigma\ \mathsf{in}\ \mathsf{let}\ v = \#1\, x\ \mathsf{in}\ \mathsf{let}\ \sigma' = \#2\ x\ \mathsf{in}\ \ldots\ .$$

Here is the translation:

$$\begin{aligned}
[\![n]\!]\,\rho\sigma &\triangleq (n, \sigma) \\
[\![x]\!]\,\rho\sigma &\triangleq (\mathsf{lookup}\ \rho\ \ulcorner x \urcorner, \sigma) \\
[\![\mathsf{if}\ e_0\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2]\!]\,\rho\sigma &\triangleq \mathsf{let}\ (b, \sigma') = [\![e_0]\!]\,\rho\sigma\ \mathsf{in} \\
&\ \ \mathsf{if}\ b\ \mathsf{then}\ [\![e_1]\!]\,\rho\sigma'\ \mathsf{else}\ [\![e_2]\!]\,\rho\sigma' \\
[\![\mathsf{ref}\ e]\!]\,\rho\sigma &\triangleq \mathsf{let}\ (v, \sigma') = [\![e]\!]\,\rho\sigma\ \mathsf{in}\ \mathsf{malloc}\ \sigma'\ v \\
[\![!\,e]\!]\,\rho\sigma &\triangleq \mathsf{let}\ (\ulcorner \ell \urcorner, \sigma') = [\![e]\!]\,\rho\sigma\ \mathsf{in}\ (\mathsf{lookup}\ \sigma'\ \ulcorner \ell \urcorner, \sigma') \\
[\![e_1 := e_2]\!]\,\rho\sigma &\triangleq \mathsf{let}\ (\ulcorner \ell \urcorner, \sigma_1) = [\![e_1]\!]\,\rho\sigma\ \mathsf{in} \\
&\ \ \mathsf{let}\ (v, \sigma_2) = [\![e_2]\!]\,\rho\sigma_1\ \mathsf{in} \\
&\ \ (\mathsf{null}, \mathsf{update}\ \sigma_2\ v\ \ulcorner \ell \urcorner) \\
[\![e_1\ ;\ e_2]\!]\,\rho\sigma &\triangleq \mathsf{let}\ (x, \sigma_1) = [\![e_1]\!]\,\rho\sigma\ \mathsf{in}\ [\![e_2]\!]\,\rho\sigma_1 \\
[\![\lambda x.\, e]\!]\,\rho_{\mathrm{lex}}\sigma_{\mathrm{lex}} &\triangleq (\lambda v\tau.\, [\![e]\!]\,(\mathsf{update}\ \rho_{\mathrm{lex}}\ v\ \ulcorner x \urcorner)\,\tau, \sigma_{\mathrm{lex}}) \\
[\![e_1 e_2]\!]\,\rho_{\mathrm{dyn}}\sigma_{\mathrm{dyn}} &\triangleq \mathsf{let}\ (f, \sigma_1) = [\![e_1]\!]\,\rho_{\mathrm{dyn}}\sigma_{\mathrm{dyn}}\ \mathsf{in} \\
&\ \ \mathsf{let}\ (v, \sigma_2) = [\![e_2]\!]\,\rho_{\mathrm{dyn}}\sigma_1\ \mathsf{in} \\
&\ \ f\, v\, \sigma_2
\end{aligned}$$

Note that the translation of abstractions $\lambda x.\, e$ take an extra argument for the store in effect at the site of the call. Thus stores are dynamically scoped.

The let construct $\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2$ can be considered syntactic sugar for $(\lambda x.\, e_2)\, e_1$, or it can be taken as primitive as in FL. If the latter, we might define

$$\begin{aligned}
[\![\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2]\!]\,\rho\sigma &\triangleq \mathsf{let}\ (v, \sigma') = [\![e_1]\!]\,\rho\sigma\ \mathsf{in} \\
&\ \ [\![e_2]\!]\,(\mathsf{update}\ \rho\ v\ \ulcorner x \urcorner)\,\sigma'.
\end{aligned}$$

Here is an example of a translation:

$$\begin{aligned}
[\![\mathsf{let}\ x = \mathsf{ref}\ 2\ \mathsf{in}\ !x]\!]\,\rho\sigma &= \mathsf{let}\ (v, \sigma_1) = [\![\mathsf{ref}\ 2]\!]\,\rho\sigma\ \mathsf{in}\ [\![!x]\!]\,(\mathsf{update}\ \rho\ v\ \ulcorner x \urcorner)\,\sigma_1 \\
&= \mathsf{let}\ (v, \sigma_1) = \mathsf{let}\ (v', \sigma_2) = [\![2]\!]\,\rho\sigma\ \mathsf{in}\ \mathsf{malloc}\ \sigma_2\ v'\ \mathsf{in} \\
&\ \ \mathsf{let}\ (\ulcorner \ell \urcorner, \sigma_3) = [\![x]\!]\,(\mathsf{update}\ \rho\ v\ \ulcorner x \urcorner)\,\sigma_1\ \mathsf{in}\ (\mathsf{lookup}\ \sigma_3\ \ulcorner \ell \urcorner, \sigma_3) \\
&= \mathsf{let}\ (v, \sigma_1) = \mathsf{let}\ (v', \sigma_2) = (2, \sigma)\ \mathsf{in}\ \mathsf{malloc}\ \sigma_2\ v'\ \mathsf{in} \\
&\ \ \mathsf{let}\ (\ulcorner \ell \urcorner, \sigma_3) = (\mathsf{lookup}\ (\mathsf{update}\ \rho\ v\ \ulcorner x \urcorner)\ \ulcorner x \urcorner, \sigma_1)\ \mathsf{in}\ (\mathsf{lookup}\ \sigma_3\ \ulcorner \ell \urcorner, \sigma_3)
\end{aligned}$$

This is a pure FL expression. Evaluating, we get

$$
\begin{aligned}
&\mathsf{let}\ (v, \sigma_1) = \mathsf{let}\ (v', \sigma_2) = (2, \sigma)\ \mathsf{in}\ \mathsf{malloc}\ \sigma_2\ v'\ \mathsf{in} \\
&\mathsf{let}\ (\ulcorner \ell \urcorner, \sigma_3) = (\mathsf{lookup}\ (\mathsf{update}\ \rho\ v\ \ulcorner x \urcorner)\ \ulcorner x \urcorner, \sigma_1)\ \mathsf{in}\ (\mathsf{lookup}\ \sigma_3\ \ulcorner \ell \urcorner, \sigma_3) \\
\to\ &\mathsf{let}\ (v, \sigma_1) = \mathsf{malloc}\ \sigma\ 2\ \mathsf{in} \\
&\mathsf{let}\ (\ulcorner \ell \urcorner, \sigma_3) = (\mathsf{lookup}\ (\mathsf{update}\ \rho\ v\ \ulcorner x \urcorner)\ \ulcorner x \urcorner, \sigma_1)\ \mathsf{in}\ (\mathsf{lookup}\ \sigma_3\ \ulcorner \ell \urcorner, \sigma_3) \\
\to\ &\mathsf{let}\ (v, \sigma_1) = (\ulcorner \ell' \urcorner, \sigma[2/\ell'])\ \mathsf{in} \\
&\mathsf{let}\ (\ulcorner \ell \urcorner, \sigma_3) = (\mathsf{lookup}\ (\mathsf{update}\ \rho\ v\ \ulcorner x \urcorner)\ \ulcorner x \urcorner, \sigma_1)\ \mathsf{in}\ (\mathsf{lookup}\ \sigma_3\ \ulcorner \ell \urcorner, \sigma_3) \\
\to\ &\mathsf{let}\ (\ulcorner \ell \urcorner, \sigma_3) = (\mathsf{lookup}\ (\mathsf{update}\ \rho\ \ell'\ \ulcorner x \urcorner)\ \ulcorner x \urcorner, \sigma[2/\ell'])\ \mathsf{in}\ (\mathsf{lookup}\ \sigma_3\ \ulcorner \ell \urcorner, \sigma_3) \\
\to\ &\mathsf{let}\ (\ulcorner \ell \urcorner, \sigma_3) = (\ulcorner \ell' \urcorner, \sigma[2/\ell'])\ \mathsf{in}\ (\mathsf{lookup}\ \sigma_3\ \ulcorner \ell \urcorner, \sigma_3) \\
\to\ &(\mathsf{lookup}\ \sigma[2/\ell']\ \ulcorner \ell' \urcorner, \sigma[2/\ell']) \\
\to\ &(2, \sigma[2/\ell']).
\end{aligned}
$$