Part VI

Imperative Functional Programming

Chapter 14

Mutable Storage

MinML is said to be a *pure* language because the execution model consists entirely of evaluating an expression for its value. ML is an *impure* language because its execution model also includes *effects*, specifically, *control effects* and *store effects*. Control effects are non-local transfers of control; these will be studied in Chapters 13 and 12. Store effects are dynamic modifications to mutable storage. This chapter is concerned with store effects.

14.1 References

The MinML type language is extended with *reference types* τ `ref` whose elements are to be thought of as mutable storage cells. We correspondingly extend the expression language with these primitive operations:

$$e \ ::= \ l \mid \texttt{ref(}e\texttt{)} \mid \texttt{!}e \mid e_1\texttt{:=}e_2$$

As in Standard ML, `ref(`e`)` allocates a "new" reference cell, `!`e retrieves the contents of the cell e, and e_1`:=`e_2 sets the contents of the cell e_1 to the value e_2. The variable l ranges over a set of *locations*, an infinite set of identifiers disjoint from variables. These are needed for the dynamic semantics, but are not expected to be notated directly by the programmer. The set of *values* is extended to include locations.

Typing judgments have the form $\Lambda; \Gamma \vdash e : \tau$, where Λ is a *location typing*, a finite function mapping locations to types; the other components of the judgement are as for MinML. The location typing Λ records the types of allocated locations during execution; this is critical for a precise statement and proof of type soundness.

The typing rules are those of MinML (extended to carry a location typing), plus the following rules governing the new constructs of the language:

$$\frac{(\Lambda(l) = \tau)}{\Lambda;\Gamma \vdash l : \tau\,\texttt{ref}} \tag{14.1}$$

$$\frac{\Lambda;\Gamma \vdash e : \tau}{\Lambda;\Gamma \vdash \texttt{ref(}e\texttt{)} : \tau\,\texttt{ref}} \tag{14.2}$$

$$\frac{\Lambda;\Gamma \vdash e : \tau\,\texttt{ref}}{\Lambda;\Gamma \vdash \texttt{!}e : \tau} \tag{14.3}$$

$$\frac{\Lambda;\Gamma \vdash e_1 : \tau_2\,\texttt{ref} \qquad \Lambda;\Gamma \vdash e_2 : \tau_2}{\Lambda;\Gamma \vdash e_1\texttt{:=}e_2 : \tau_2} \tag{14.4}$$

Notice that the location typing is not extended during type checking! Locations arise only during execution, and are not part of complete programs, which must not have any free locations in them. The role of the location typing will become apparent in the proof of type safety for MinML extended with references.

A *memory* is a finite function mapping locations to closed values (but possibly involving locations). The dynamic semantics of MinML with references is given by an abstract machine. The states of this machine have the form (M, e), where M is a memory and e is an expression possibly involving free locations in the domain of M. The locations in $\mathrm{dom}(M)$ are bound simultaneously in (M, e); the names of locations may be changed at will without changing the identity of the state.

The transitions for this machine are similar to those of the M machine, but with these additional steps:

$$\frac{(M, e) \mapsto (M', e')}{(M, \texttt{ref(}e\texttt{)}) \mapsto (M, \texttt{ref(}e'\texttt{)})} \tag{14.5}$$

$$\frac{(l \notin \mathrm{dom}(M))}{(M, \texttt{ref(}v\texttt{)}) \mapsto (M[l{=}v], l)} \tag{14.6}$$

$$\frac{(M, e) \mapsto (M', e')}{(M, \texttt{!}e) \mapsto (M', \texttt{!}e')} \tag{14.7}$$

$$\frac{(l \in \mathrm{dom}(M))}{(M, \,!\,l) \mapsto (M, M(l))} \tag{14.8}$$

$$\frac{(M, e_1) \mapsto (M', e_1')}{(M, e_1 \,\mathsf{:=}\, e_2) \mapsto (M', e_1' \,\mathsf{:=}\, e_2)} \tag{14.9}$$

$$\frac{(M, e_2) \mapsto (M', e_2')}{(M, v_1 \,\mathsf{:=}\, e_2) \mapsto (M', v_1 \,\mathsf{:=}\, e_2')} \tag{14.10}$$

$$\frac{(l \in \mathrm{dom}(M))}{(M, l \,\mathsf{:=}\, v) \mapsto (M[l{=}v], v)} \tag{14.11}$$

A state (M, e) is *final* iff e is a value (possibly a location).

To prove type safety for this extension we will make use of some auxiliary relations. Most importantly, the typing relation between memories and location typings, written $\vdash M : \Lambda$, is inductively defined by the following rule:

$$\frac{\mathrm{dom}(M) = \mathrm{dom}(\Lambda) \quad \forall l \in \mathrm{dom}(\Lambda)\ \Lambda; \bullet \vdash M(l) : \Lambda(l)}{\vdash M : \Lambda} \tag{14.12}$$

It is very important to study this rule carefully! First, we require that Λ and M govern the same set of locations. Second, for each location l in their common domain, we require that the value at location l, namely $M(l)$, have the type assigned to l, namely $\Lambda(l)$, relative to the *entire* location typing Λ. This means, in particular, that memories may be "circular" in the sense that the value at location l may contain an occurrence of l, for example if that value is a function.

The typing rule for memories is reminiscent of the typing rule for recursive functions — we are allowed to assume the typing that we are trying to prove while trying to prove it. This similarity is no accident, as the following example shows. Here we use ML notation, but the example can be readily translated into MinML extended with references:

```
(* loop forever when called *)
fun diverge (x:int):int = diverge x
(* allocate a reference cell *)
val fc : (int->int) ref = ref (diverge)
(* define a function that ''recurs'' through fc *)
fun f 0 = 1 | f n = n * ((!fc)(n-1))
(* tie the knot *)
val _ = fc := f
(* now call f *)
val n = f 5
```

This technique is called *backpatching*. It is used in some compilers to implement recursive functions (and other forms of looping construct).

Exercise 34

1. *Sketch the contents of the memory after each step in the above example. Observe that after the assignment to* `fc` *the memory is "circular" in the sense that some location contains a reference to itself.*

2. *Prove that every cycle in well-formed memory must "pass through" a function. Suppose that* $M(l_1) = l_2$, $M(l_2) = l_3$, *...*, $M(l_n) = l_1$ *for some sequence* $l_1, \ldots, l_n$ *of locations. Show that there is no location typing* Λ *such that* $\vdash M : \Lambda$.

The well-formedness of a machine state is inductively defined by the following rule:

$$\frac{\vdash M : \Lambda \qquad \Lambda; \bullet \vdash e : \tau}{(M, e)\ \mathsf{ok}} \tag{14.13}$$

That is, (M, e) is well-formed iff there is a location typing for M relative to which e is well-typed.

Theorem 35 (Preservation)
If (M, e) ok *and* $(M, e) \mapsto (M', e')$, *then* (M', e') ok.

Proof: The trick is to prove a stronger result by induction on evaluation: if $(M, e) \mapsto (M', e')$, $\vdash M : \Lambda$, and $\Lambda; \bullet \vdash e : \tau$, then there exists $\Lambda' \supseteq \Lambda$ such that $\vdash M' : \Lambda'$ and $\Lambda'; \bullet \vdash e' : \tau$. ■

Exercise 36
Prove Theorem 35. The strengthened form tells us that the location typing, and the memory, increase monotonically during evaluation — the type of a

location never changes once it is established at the point of allocation. This is crucial for the induction.

Theorem 37 (Progress)
If (M, e) ok *then either* (M, e) *is a final state or there exists* (M', e') *such that* $(M, e) \mapsto (M', e')$.

Proof: The proof is by induction on typing: if $\vdash M : \Lambda$ and $\Lambda; \bullet \vdash e : \tau$, then either e is a value or there exists $M' \supseteq M$ and e' such that $(M, e) \mapsto (M', e')$. ■

Exercise 38
Prove Theorem 37 by induction on typing of machine states.

Chapter 15

Monads

As we saw in Chapter 14 one way to combine functional and imperative programming is to add a type of reference cells to MinML. This approach works well for call-by-value languages,[1] because we can easily predict where expressions are evaluated, and hence where references are allocated and assigned. For call-by-name languages this approach is problematic, because in such languages it is much harder to predict when (and how often) expressions are evaluated.

Enriching ML with a type of references has an additional consequence that one can no longer determine from the type alone whether an expression mutates storage. For example, a function of type `int`→`int` must taken an integer as argument and yield an integer as result, but may or may not allocate new reference cells or mutate existing reference cells. The expressive power of the type system is thereby weakened, because we cannot distinguish *pure* (effect-free) expressions from *impure* (effect-ful) expressions.

Another approach to introducing effects in a purely functional language is to make the use of effects explicit in the type system. Several methods have been proposed, but the most elegant and widely used is the concept of a *monad*. Roughly speaking, we distinguish between *pure* and *impure* expressions, and make a corresponding distinction between *pure* and *impure* function types. Then a function of type `int`→`int` is a pure function (has no effects when evaluated), whereas a function of type `int` ⇀ `int` may have an effect when applied. The monadic approach is more popular for call-by-name languages, but is equally sensible for call-by-value languages.

[1] We need to introduce cbv and cbn earlier, say in Chapter 8.

15.1 Monadic MinML

A monadic variant of MinML is obtained by separating pure from impure expressions. The pure expressions are those of MinML. The impure expressions consist of any pure expression (vacuously impure), plus a new primitive expression, called *bind*, for sequencing evaluation of impure expressions. In addition the impure expressions include primitives for allocating, mutating, and accessing storage; these are "impure" because they depend on the store for their execution.

The abstract syntax of monadic MinML is given by the following grammar:

$$
\begin{array}{llll}
\textit{Types} & \tau & ::= & \texttt{int} \mid \texttt{bool} \mid \tau_1 \rightarrow \tau_2 \mid \tau_1 \rightharpoonup \tau_2 \\
\textit{Pure} & e & ::= & x \mid n \mid o(e_1, \ldots, e_n) \mid \\
 & & & \texttt{true} \mid \texttt{false} \mid \texttt{if}\, e\, \texttt{then}\, e_1\, \texttt{else}\, e_2\, \texttt{fi} \mid \\
 & & & \texttt{fun}\, f\, (x{:}\tau_1){:}\tau_2\, \texttt{is}\, e\, \texttt{end} \mid \texttt{apply}(e_1, e_2) \\
 & & & \texttt{fun}\, f\, (x{:}\tau_1){:}\tau_2\, \texttt{is}\, m\, \texttt{end} \\
\textit{Impure} & m & ::= & \texttt{return}\, e \mid \texttt{bind}\, x{:}\tau \leftarrow m_1\, \texttt{in}\, m_2 \\
 & & & \texttt{if}_\tau\, e\, \texttt{then}\, m_1\, \texttt{else}\, m_2\, \texttt{fi} \mid \texttt{apply}(e_1, e_2)
\end{array}
$$

Monadic MinML is a general framework for computing with effects. Note that there are two forms of function, one whose body is pure, and one whose body is impure. Correspondingly, there are two forms of application, one for pure functions, one for impure functions. There are also two forms of conditional, according to whether the arms are pure or impure. (We will discuss methods for eliminating some of this redundancy below.)

The static semantics of monadic MinML consists of two typing judgements, $\Gamma \vdash e : \tau$ for pure expressions, and $\Gamma \vdash m : \tau$ for impure expressions.

Most of the rules are as for MinML; the main differences are given below.

$$\frac{\Gamma, f{:}\tau_1 \rightharpoonup \tau_2, x{:}\tau_1 \vdash m : \tau_2}{\Gamma \vdash \texttt{fun}\, f\, \texttt{(}x\texttt{:}\tau_1\texttt{):}\tau_2\, \texttt{is}\, m\, \texttt{end} : \tau_1 \rightharpoonup \tau_2}$$

$$\frac{\Gamma \vdash e_1 : \tau_2 \rightharpoonup \tau \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash \texttt{apply(}:, \tau\texttt{)}}$$

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash \texttt{return}\, e : \tau}$$

$$\frac{\Gamma \vdash m_1 : \tau_1 \quad \Gamma, x{:}\tau_1 \vdash m_2 : \tau_2}{\Gamma \vdash \texttt{bind}\, x{:}\tau \leftarrow m_1\, \texttt{in}\, m_2 : \tau_2}$$

$$\frac{\Gamma \vdash e : \texttt{bool} \quad \Gamma \vdash m_1 : \tau \quad \Gamma \vdash m_2 : \tau}{\Gamma \vdash \texttt{if}_\tau\, e\, \texttt{then}\, m_1\, \texttt{else}\, m_2\, \texttt{fi} : \tau}$$

So far we have not presented any mechanisms for engendering effects! Monadic MinML is rather a framework for a wide variety of effects that we will instantiate to the case of mutable storage. This is achieved by adding the following forms of impure expression to the language:

$$\textit{Impure} \quad m \quad ::= \quad \texttt{ref(}e\texttt{)} \mid \texttt{!}e \mid e_1\texttt{:=}e_2$$

Their typing rules are as follows:

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash \texttt{ref(}e\texttt{)} : \tau\, \texttt{ref}}$$

$$\frac{\Gamma \vdash e : \tau\, \texttt{ref}}{\Gamma \vdash \texttt{!}e : \tau}$$

$$\frac{\Gamma \vdash e_1 : \tau\, \texttt{ref} \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1\texttt{:=}e_2 : \tau_2}$$

In addition we include locations as pure expressions, with typing rule

$$\frac{(\Gamma(l) = \tau)}{\Gamma \vdash l : \tau\, \texttt{ref}}$$

(For convenience we merge the location and variable typings.)

The dynamic semantics of monadic MinML is an extension to that of MinML. Evaluation of pure expressions does not change, but we must

add rules governing evaluation of impure expressions. For the purposes of describing mutable storage, we must consider transitions of the form $(M, m) \mapsto (M', m')$, where M and M' are memories, as in Chapter 14.

$$\frac{e \mapsto e'}{(M, \mathtt{return}\, e) \mapsto (M, \mathtt{return}\, e')}$$

$$\frac{(M, m_1) \mapsto (M', m_1')}{(M, \mathtt{bind}\, x{:}\tau \leftarrow m_1\, \mathtt{in}\, m_2) \mapsto (M', \mathtt{bind}\, x{:}\tau \leftarrow m_1'\, \mathtt{in}\, m_2)}$$

$$\frac{}{(M, \mathtt{bind}\, x{:}\tau \leftarrow \mathtt{return}\, v\, \mathtt{in}\, m_2) \mapsto (M, \{v/x\}m_2)}$$

The evaluation rules for the reference primitives are as in Chapter 14.

15.2 Reifying Effects

The need for pure and impure function spaces in monadic MinML is somewhat unpleasant because of the duplication of constructs. One way to avoid this is to introduce a new type constructor, $!\tau$, whose elements are unevaluated impure expressions. The computation embodied by the expression is said to be *reified* (turned into a "thing").

The syntax required for this extension is as follows:

Types	τ	$::=$	$!\tau$
Pure	e	$::=$	$\mathsf{box}(m)$
Impure	m	$::=$	$\mathsf{unbox}(e)$

Informally, the pure expression $\mathsf{box}(m)$ is a value that contains an *unevaluated* impure expression m; the expression m is said to be *boxed*. Boxed expressions can be used as ordinary values without restriction. The expression $\mathsf{unbox}(e)$ "opens the box" and evaluates the impure expression inside; it is therefore itself an impure expression.

The static semantics of this extension is given by the following rules:

$$\frac{\Gamma \vdash m : \tau}{\Gamma \vdash \mathsf{box}(m) : !\tau}$$

$$\frac{\Gamma \vdash e : !\tau}{\Gamma \vdash \mathsf{unbox}(e) : \tau}$$

The dynamic semantics is given by the following transition rules:

$$\frac{}{(M, \mathtt{unbox}(\mathtt{box}(m))) \mapsto (M, m)}$$
$$\frac{e \mapsto e'}{(M, \mathtt{unbox}(e)) \mapsto (M, \mathtt{unbox}(e'))}$$

The expression $\mathtt{box}(m)$ is a value, for any choice of m.

One use for reifying effects is to replace the impure function space, $\tau_1 \rightharpoonup \tau_2$, with the pure function space $\tau_1 \rightarrow ! \, \tau_2$. The idea is that an impure function is a pure function that yields a suspended computation that must be unboxed to be executed. The impure function expression

$$\mathtt{fun}\, f\, \mathtt{(}x \mathtt{:} \tau_1\mathtt{)} \mathtt{:} \tau_2\, \mathtt{is}\, m\, \mathtt{end}$$

is replaced by the pure function expression

$$\mathtt{fun}\, f\, \mathtt{(}x \mathtt{:} \tau_1\mathtt{)} \mathtt{:} \tau_2\, \mathtt{is}\, \mathtt{box}(m)\, \mathtt{end}.$$

The impure application,

$$\mathtt{apply(}e_1, e_2\mathtt{)},$$

is replaced by

$$\mathtt{unbox}(\mathtt{apply(}e_1, e_2\mathtt{)}),$$

which unboxes, hence executes, the suspended computation.

15.3 Exercises

1. Consider other forms of effect such as I/O.
2. Check type safety.
3. Problems with multiple monads to distinguish multiple effects.