Lecture Notes on Mutable Storage

15-312: Foundations of Programming Languages
Frank Pfenning

Lecture 16
October 26, 2004

After several lectures on extensions to the type system that are independent from computational mechanism, we now consider mutable storage as a computational effect. This is a counterpart to the study of exceptions and continuations which are *control effects*.

We will look at mutable storage from two different points of view: one, where essentially all of MinML becomes an imperative language and one where we use the type system to isolate effects (next lecture). The former approach is taken in ML, that latter in Haskell.

To add effects in the style of ML, we add a new type $\tau\,\mathsf{ref}$ and three new expressions to create a mutable cell ($\mathsf{ref}(e)$), to write to the cell ($e_1 := e_2$), and read the contents of the cell ($!e$). There is only a small deviation from the semantics of Standard ML here in that updating a cell returns its new value instead of the unit element. We also need to introduce cell labels themselves so we can uniquely identify them. We write l for *locations*. Locations are assigned types in a *store typing* Λ.

$$\textit{Store Typings} \qquad \Lambda ::= \cdot \mid \Lambda, l{:}\tau$$

Locations never appear in the input program, but they can arise during evaluation, when cells are allocated using the $\mathsf{ref}(e)$ construct. We therefore need to thread the store typing through the typing judgment which now has the form $\Lambda; \Gamma \vdash e : \tau$. We obtain the following rules, which should be familiar from Standard ML. We use here the concrete, rather than the abstract syntax, in order to present the assignment and dereferencing operations.

$$\frac{\Lambda;\Gamma \vdash e : \tau}{\Lambda;\Gamma \vdash \mathsf{ref}(e) : \tau\ \mathsf{ref}} \qquad \frac{\Lambda;\Gamma \vdash e_1 : \tau\ \mathsf{ref} \quad \Lambda;\Gamma \vdash e_2 : \tau}{\Lambda;\Gamma \vdash e_1 := e_2 : \tau}$$

$$\frac{\Lambda;\Gamma \vdash e : \tau\ \mathsf{ref}}{\Lambda;\Gamma \vdash\ !e : \tau} \qquad \frac{l{:}\tau \text{ in } \Lambda}{\Lambda;\Gamma \vdash \mathsf{loc}(l) : \tau\ \mathsf{ref}}$$

It is important to keep in mind the difference between locations and variables. Expressions that we evaluate are always closed with respect to variables (we substitute for them), but they may contain references l to locations.

To describe the operational semantics, we need to model the store. We think of it simply as a mapping from locations to values and we denote it by M for memory.

$$\textit{Stores} \qquad M ::= \cdot \mid M, l{=}v$$

Note that in the evaluation of a functional program in a real compiler there are many other uses of memory (heap and stack, for example), while the store only contains the mutable cells. As usual, we assume that all locations in a store are distinct.

In this approach to modeling mutable storage, the evaluation of any expression can potentially have an effect. This means we need to change our basic model of computation to add a store. We replace the ordinary transition judgment $e \mapsto e'$ by

$$\langle M, e\rangle \mapsto \langle M', e'\rangle$$

which asserts that expression e in store M steps to expression e' with store M'. First, we have to take care of changing *all* prior rules to thread through the store. Fortunately, this is quite systematic. We show only the cases for functions.

$$\frac{\langle M, e_1\rangle \mapsto \langle M', e_1'\rangle}{\langle M, \mathtt{apply}(e_1, e_2)\rangle \mapsto \langle M', \mathtt{apply}(e_1', e_2)\rangle}$$

$$\frac{v_1\ \mathsf{value} \quad \langle M, e_2\rangle \mapsto \langle M', e_2'\rangle}{\langle M, \mathtt{apply}(v_1, e_2)\rangle \mapsto \langle M', \mathtt{apply}(v_1, e_2')\rangle}$$

$$\frac{v_2\ \mathsf{value}}{\langle M, \mathtt{apply}(\mathtt{fn}(\tau_2, x.e), v_2)\rangle \mapsto \langle M, \{v_2/x\}e\rangle}$$

For the new operations we have to be careful about the evaluation order, and also take into account that evaluating, say, the initializer of a new cell may actually change the store.

$$\frac{\langle M, e\rangle \mapsto \langle M', e'\rangle}{\langle M, \mathsf{ref}(e)\rangle \mapsto \langle M', \mathsf{ref}(e')\rangle} \quad \frac{v\ \ \mathsf{value}}{\langle M, \mathsf{ref}(v)\rangle \mapsto \langle (M, l{=}v), \mathsf{loc}(l)\rangle} \quad \frac{}{\mathsf{loc}(l)\ \ \mathsf{value}}$$

$$\frac{\langle M, e_1\rangle \mapsto \langle M', e_1'\rangle}{\langle M, e_1 := e_2\rangle \mapsto \langle M', e_1' := e_2\rangle} \qquad \frac{v_1\ \ \mathsf{value} \quad \langle M, e_2\rangle \mapsto \langle M', e_2'\rangle}{\langle M, v_1 := e_2\rangle \mapsto \langle M', v_1 := e_2'\rangle}$$

$$\frac{M = (M_1, l{=}v_1, M_2) \quad \text{and} \quad M' = (M_1, l{=}v_2, M_2)}{\langle M, \mathsf{loc}(l) := v_2\rangle \mapsto \langle M', v_2\rangle}$$

$$\frac{\langle M, e\rangle \mapsto \langle M', e'\rangle}{\langle M, !e\rangle \mapsto \langle M', !e'\rangle} \qquad \frac{M = (M_1, l{=}v, M_2)}{\langle M, !\mathsf{loc}(l)\rangle \mapsto \langle M, v\rangle}$$

In order to state type preservation and progress we need to define well-formed machine states which in turn requires validity for the memory configuration. For that, we need to check that each cell contains a value of the type prescribed by the store typing. The value stored in each cell can reference other cells which can in turn refer back to the first cell. In other words, the pointer structure of memory can be cyclic. We therefore need to check the contents of each cell knowing the typing of all locations. The judgment has the form $\Lambda_0; \cdot \vdash M : \Lambda$, where we intend Λ_0 to range over the whole store typing will we verify on the right-hand side that each cell has the prescribed type.

$$\frac{}{\Lambda_0; \cdot \vdash (\cdot) : (\cdot)} \qquad \frac{\Lambda_0; \cdot \vdash M : \Lambda \quad \Lambda_0; \cdot \vdash v : \tau \quad v\ \ \mathsf{value}}{\Lambda_0; \cdot \vdash (M, l{=}v) : (\Lambda, l{:}\tau)}$$

With this defined, we can state appropriate forms of type preservation and progress theorems. We write $\Lambda' \geq \Lambda$ if Λ' is an extension of the store typing Λ with some additional locations. In this particular case, for a single step, we need at most one new location so that if $\Lambda' \geq \Lambda$ then either $\Lambda' = \Lambda$ or $\Lambda' = \Lambda, l{:}\tau$ for a new l and some τ.

Theorem 1 (Type Preservation)
If $\Lambda; \cdot \vdash e : \tau$ and $\Lambda; \cdot \vdash M : \Lambda$ and $\langle M, e\rangle \mapsto \langle M', e'\rangle$ then for some $\Lambda' \geq \Lambda$ and memory M' we have $\Lambda'; \cdot \vdash e' : \tau$ and $\Lambda'; \cdot \vdash M' : \Lambda'$.

Proof: By induction on the derivation of the computation judgment, applying inversion on the typing assumptions. ■

Theorem 2 (Progress)
If $\Lambda; \cdot \vdash e : \tau$ and $\Lambda; \cdot \vdash M : \Lambda$ then either

(i) e value*, or*

(ii) $\langle M, e \rangle \mapsto \langle M', e' \rangle$ *for some M' and e'.*

Proof: By induction on the derivation of the typing judgment, analyzing all possible cases. ■

We assume the reader is already familiar with the usual programming idioms using references and assignment. As an example that illustrates one of the difficulties of reasoning about programs with possibly hidden effect, consider the following ML code.

```
signature COUNTER =
sig
  type c
  val new : int -> c  (* create a counter *)
  val inc : c -> int  (* inc and return new value *)
end;
structure C :> COUNTER =
struct
  type c = int ref
  fun new(n):c = ref(n)
  fun inc(r) = (r := !r+1; !r)
end;
val c = C.new(0);
val 1 = C.inc(c);
val 2 = C.inc(c);
```

Here the two calls to `C.inc(c)` are identical but yield different results. This is the intended behavior, but clearly not exposed in the type of the expressions involved. There are many pitfalls in programming with ephemeral data structures that most programmers are all too familiar with.

The way we have extended MinML with mutable storage has several drawbacks. The principal difficulty with programming with effects is that the type system does not track them properly. So when we examine the type of a function $\tau_1 \to \tau_2$ we cannot tell if the function simply returns a value of type τ_2 or if it could also have an effect. This complicates reasoning about programs and their correctness tremendously.

An alternative is to try to express in the type system that certain functions may have effects, while others do not have effects. This is the purpose of *monads* that are quite popular in the Haskell community. Haskell is a lazy[1] functional language in which all effects are isolated in a monad. We will see that monadic programming has its own drawbacks. The last word in the debate on how to integrate imperative and pure functional programming has not yet been spoken.

We introduce monads in two steps. The first step is the generic framework, which can be instantiated to different kinds of effects. In this lecture we introduce mutable storage as an effect, just as we did in the previous lecture on mutable storage in ML.

In the generic framework, we extend MinML by adding a new syntactic category of *monadic expressions*, denoted by m.[2] Correspondingly, there is a new typing judgment

$$\Gamma \vdash m \div \tau$$

expressing that the monadic expression m has type τ in context Γ. We think of a monadic expression as one whose evaluation returns not only a value of type τ, but also may have an effect. We introduce this separate category so that the ordinary expressions we have used so far can remain pure, that is, free of effects.

Any particular use of the monadic framework will add particular new monadic expressions, and also possibly new pure expressions. But first the constructs that are independent of the kind of effect we want to consider. The first principle is that a pure expression e can be considered as a monadic expression e which happens to have no effect.

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash e \div \tau}$$

The second idea is that we can quote a monadic expression and thereby turn it into a pure expression. It has no effects because the monadic expression will not be executed. We write the quotation operator as $\mathsf{comp}(m)$, and the type of quoted computations of type τ as $\bigcirc\tau$.

$$\frac{\Gamma \vdash m \div \tau}{\Gamma \vdash \mathsf{comp}(m) : \bigcirc\tau} \qquad \frac{}{\mathsf{comp}(m)\ \ \mathsf{value}}$$

[1] Lazy here means call-by-name with memoization of the suspension.

[2] In lecture, we did not use a separate syntactic category, but just as writing v for values aids understanding, writing m for potentially effectful expressions makes it easier to interpret some rules.

Finally, we must be able to unwrap and thereby actually execute a quoted monadic expression. However, we cannot do this anywhere in a pure expression, because evaluating such a supposedly pure expression would then have an effect. Instead, we can only do this if we are within an explicit sequence of monadic expressions! This yields the following construct

$$\frac{\Gamma \vdash e : \bigcirc\tau \quad \Gamma, x{:}\tau \vdash m \div \sigma}{\Gamma \vdash \mathsf{letcomp}(e, x.\, m) \div \sigma}$$

We will use the concrete syntax $\mathsf{let\,comp}\, x = e \,\mathsf{in}\, m \,\mathsf{end}$ for $\mathsf{letcomp}(e, x.\, m)$. Note that m and $\mathsf{letcomp}(e, x.\, m)$ are monadic expressions and therefore may have an effect, while e is a pure expression of monadic type. We think of the effects are being staged as follows:

(1) We evaluate e which should yield a value $\mathsf{comp}(m')$.

(2) We execute the monadic expression m', which will have some effects but also return a value v.

(3) Substitute v for x in m and then execute the resulting monadic expression.

In order to specify this properly we need to be able to describe the effect that may be engendered by executing a monadic expression. The judgment for executing monadic expressions then has the form

$$\langle M, m \rangle \mapsto \langle M', m' \rangle$$

where the store changes from M to M' and the expression steps from m to m'. According to the considerations above, we obtain first the following rules, where we use a pure expression as a (trivial) form of monadic expression.

$$\frac{e\ \mathsf{pure} \quad e \mapsto e'}{\langle M, e \rangle \mapsto \langle M, e' \rangle}$$

Here we have used $e\ \mathsf{pure}$ for the judgment that e can be classified by the typing rules as $e : \tau$. Just like the property of being a value, this is a purely syntactic property of e. Furthermore, it is shallow: $\mathsf{letcomp}$, allocation, assignment, and dereference are monadic expressions while all others are pure.

We can see that the transition judgment on ordinary expressions looks the same as before and that it can have no effect. Contrast this with the situation in ML from the previous lecture where we needed to change *every* transition rule to account for possible effects.

The next sequence of three rules implement items (1), (2), and (3) above.

$$\frac{e \mapsto e'}{\langle M, \mathsf{letcomp}(e, x.\, m)\rangle \mapsto \langle M, \mathsf{letcomp}(e', x.\, m)\rangle}$$

$$\frac{\langle M, m_1\rangle \mapsto \langle M', m_1'\rangle}{\langle M, \mathsf{letcomp}(\mathsf{comp}(m_1), x.\, m)\rangle \mapsto \langle M', \mathsf{letcomp}(\mathsf{comp}(m_1'), x.\, m)\rangle}$$

$$\frac{v\ \ \mathsf{value}}{\langle M, \mathsf{letcomp}(\mathsf{comp}(v), x.\, m)\rangle \mapsto \langle M, \{v/x\}m\rangle}$$

Note that the substitution in the last rule is appropriate. The substitution principle for pure values into monadic expressions is straightforward precisely because v is cannot have effects.

We will not state here the generic forms of the preservation and progress theorems. They are somewhat trivialized because our language, while designed with effects in mind, does not yet have any actual effects.

In order to define the monad for mutable storage we introduce a new form of type, $\tau\,\mathsf{ref}$ and three new forms of monadic expressions, namely $\mathsf{ref}(e)$, $e_1 := e_2$ and $!e$. In addition we need one new form of pure expression, namely locations l which are declared in a store typing Λ with their type. Recall the form of store typings.

$$\textit{Store Typings} \qquad \Lambda ::= \cdot \mid \Lambda, l{:}\tau$$

Locations can be pure because creating, assigning, or dereferencing them is an effect, and the types prevent any other operations on them. The store typing must now be taking into account when checking expressions that are created a runtime. They are, however, not needed for compile-time checking because the program itself, before it is started, cannot directly refer to locations. We just uniformly add "Λ;" to all the typing judgments—they are simply additional hypotheses of a slightly different form than what is recorded in Γ.

$$\frac{\Lambda; \Gamma \vdash e : \tau}{\Lambda; \Gamma \vdash \mathsf{ref}(e) \div \tau\,\mathsf{ref}} \qquad \frac{\Lambda; \Gamma \vdash e_1 : \tau\,\mathsf{ref} \quad \Lambda; \Gamma \vdash e_2 : \tau}{\Lambda; \Gamma \vdash e_1 := e_2 \div \tau}$$

$$\frac{\Lambda; \Gamma \vdash e : \tau\,\mathsf{ref}}{\Lambda; \Gamma \vdash\, !e \div \tau} \qquad \frac{l{:}\tau \text{ in } \Lambda}{\Lambda; \Gamma \vdash \mathsf{loc}(l) : \tau\,\mathsf{ref}}$$

Note that the constituents of the new monadic expressions are pure expressions. This guarantees that they cannot have effects: all effects must be explicitly sequenced using the letcomp form.

Now the additional rules for new expressions are analogous to those we had when effects where not encapsulated in the monad.

$$\frac{e \mapsto e'}{\langle M, \mathsf{ref}(e)\rangle \mapsto \langle M, \mathsf{ref}(e')\rangle} \qquad \frac{v \;\mathsf{value}}{\langle M, \mathsf{ref}(v)\rangle \mapsto \langle (M, l{=}v), \mathsf{loc}(l)\rangle} \qquad \frac{}{l \;\mathsf{value}}$$

$$\frac{e_1 \mapsto e_1'}{\langle M, e_1 := e_2\rangle \mapsto \langle M, e_1' := e_2\rangle} \qquad \frac{v_1 \;\mathsf{value} \quad e_2 \mapsto e_2'}{\langle M, v_1 := e_2\rangle \mapsto \langle M, v_1 := e_2'\rangle}$$

$$\frac{M = (M_1, l{=}v_1, M_2) \quad \mathsf{and} \quad M' = (M_1, l{=}v_2, M_2) \quad v_2 \;\mathsf{value}}{\langle M, \mathsf{loc}(l) := v_2\rangle \mapsto \langle M', v_2\rangle}$$

$$\frac{e \mapsto e'}{\langle M, !e\rangle \mapsto \langle M, !e'\rangle} \qquad \frac{M = (M_1, l{=}v, M_2)}{\langle M, !\mathsf{loc}(l)\rangle \mapsto \langle M, v\rangle}$$

We complete this lecture with a simple example. In the next lecture we discuss the progress and preservation properties and other forms of effects. In MinML with pervasive effects, we might write the following, which allocates a cell initialized with 3 and then increments it.

```
let x = ref 3
in x := !x + 1 end;
```

In MinML with effects encapsulated in a monad, we would rewrite this as follows.

```
let comp x = comp (ref 3) in
let comp y = comp (!x) in
let comp z = comp (x := y+1) in
z end end end
```

Note that the arguments to assignment must be pure expressions, so we must explicitly sequence the computation into two assignments.

It is this rewriting of expressions which is often required that can make programming with effects in monadic style tedious (although some syntactic sugar can clearly help). Another problem is that operations such as input/output are also effects and therefore must be inside the monad.

This means that inserting a print statement into a function changes its type, which can complicate debugging.