# Supplementary Notes on Mutable Storage

15-312: Foundations of Programming Languages
Frank Pfenning

Lecture 14
October 10, 2002

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* [Ch. 14].

We will look at mutable storage from two different points of view: one, where essentially all of MinML becomes an imperative language (this lecture), 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* $\Lambda$.

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

Since locations can be mentioned anywhere in a program, we 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 ML.

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

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

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.

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.

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

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

$$\frac{(v_1 = \mathtt{fun}(\tau_1, \tau_2, f.x.e)) \quad v_2\ \mathsf{value}}{\langle M, \mathtt{apply}(v_1, v_2)\rangle \mapsto \langle M, \{v_1/f\}\{v_2/x\}e\rangle}$$

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.

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

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

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

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

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 refer other cells which can in turn refer back to the first cell. In other words, the pointer structure 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 $\Lambda_0$ 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 $\Lambda'$ is an extension of the store typing $\Lambda$ with some additional locations. In this particular case, for a single step, we need at most one new location.

**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)* $e$ $\mathsf{value}$*, 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 too familiar with.