15–212: Fundamental Structures of Computer Science II

Some Notes on Mutable References

Frank Pfenning

Draft of October 18, 1997

These notes provide a brief introduction to the evaluation model underlying mutable references and arrays. We assume that the reader is already familiar with ML and the *Notes on Evaluation* handed out earlier in this class.

1 Notation

Recall that e stands for arbitrary expressions in ML and v for values, which are a special kind of expression. We wrote

$e \hookrightarrow v$	expression e *evaluates to* value v
$e \stackrel{1}{\Longrightarrow} e'$	expression e *reduces to* e' in 1 step
$e \stackrel{k}{\Longrightarrow} e'$	expression e *reduces to* e' in k steps
$e \Longrightarrow e'$	expression e *reduces to* e' in 0 or more steps

Our notion of *step* in the operational semantics is defined abstractly and does not coincide with the actual operations performed in an implementation of ML. Since we will be mainly concerned with proving correctness, but not complexity of implementation, the number of steps is largely irrelevant and we will write $e \Longrightarrow e'$ for reduction.

Evaluation and reduction are related in the sense that if $e \hookrightarrow v$ then $e \stackrel{1}{\Longrightarrow} e_1 \stackrel{1}{\Longrightarrow} \cdots \stackrel{1}{\Longrightarrow} v$ and *vice versa*.

Note that values evaluate to themselves "in 0 steps". In particular, for a value v there is no expression e such that $v \stackrel{1}{\Longrightarrow} e$.

2 Store

A notion of mutable reference introduces a *store* into the operational semantics. Under this extension, expressions not only have a value (or fail to have a value if they do not terminate), but they may now also have an *effect* on the store. Other effects we ignore here are exceptions (introduced informally in class) and input/output.

A store is modelled as a collection of *cells*, each with a unique label c. Cells are typed, and each cell contains a value which matches its type. We write $c \mapsto v$ if the contents of the cell c is v, and we write a store s as

$$s = c_1 \mapsto v_1, \ldots, c_n \mapsto v_n$$

with the invariant that all c_i are distinct.

The operational semantics now relates pairs $\langle s; e\rangle$ consisting of the store s and the expression e to be evaluated. We write

$$\begin{array}{ll} \langle s; e\rangle \stackrel{1}{\Longrightarrow} \langle s'; e'\rangle & \text{expression } e \textit{ reduces to } e' \text{ in 1 step, transforming store } s \text{ to } s' \\ \langle s; e\rangle \stackrel{k}{\Longrightarrow} \langle s'; e'\rangle & \text{expression } e \textit{ reduces to } e' \text{ in } k \text{ steps, transforming store } s \text{ to } s' \\ \langle s; e\rangle \Longrightarrow \langle s'; e'\rangle & \text{expression } e \textit{ reduces to } e' \text{ in 0 or more steps, transforming store } s \text{ to } s' \end{array}$$

The rule we introduced previously have to be modified in a systematic way to account for the store. As an example, we show the rules for Booleans.

$$\begin{array}{rcll} \texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3 & \stackrel{1}{\Longrightarrow} & \texttt{if } e_1' \texttt{ then } e_2 \texttt{ else } e_3 & \text{if } e_1 \stackrel{1}{\Longrightarrow} e_1' \\ \texttt{if true then } e_2 \texttt{ else } e_3 & \stackrel{1}{\Longrightarrow} & e_2 & \\ \texttt{if false then } e_2 \texttt{ else } e_3 & \stackrel{1}{\Longrightarrow} & e_3 & \end{array}$$

These now read

$$\begin{array}{rcll} \langle s; \texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3\rangle & \stackrel{1}{\Longrightarrow} & \langle s'; \texttt{if } e_1' \texttt{ then } e_2 \texttt{ else } e_3\rangle & \text{if } \langle s; e_1\rangle \stackrel{1}{\Longrightarrow} \langle s'; e_1'\rangle \\ \langle s; \texttt{if true then } e_2 \texttt{ else } e_3\rangle & \stackrel{1}{\Longrightarrow} & \langle s; e_2\rangle & \\ \langle s; \texttt{if false then } e_2 \texttt{ else } e_3\rangle & \stackrel{1}{\Longrightarrow} & \langle s; e_3\rangle & \end{array}$$

It should be clear from these how all the other rules should be modified.

3 Mutable References

We have a new type constructor `ref`.

Types. t `ref` for any type t. This is the type of cells with contents of type t.

Values. The only new values are the cells c, identified by their label. The label itself is not directly accessible in ML, because it is unpredictable which label might be chosen for a freshly created cell. Instead a value c with $c \mapsto v$ in the current store is printed as `ref` v. This is potentially confusing, because two different cells with the same contents are printed the same way, even though they are different. Always keep this in mind when interpreting output from ML's top-level!

Operations. We have operations to create a new cell (`ref` e), to read the contents of a cell (`!`e), and to update the contents of a cell (e_1 `:=` e_2).

Typing Rules.

$\texttt{ref } e : t \texttt{ ref}$ if $e : t$

$\texttt{!}e : t$ if $e : t \texttt{ ref}$

$e_1 \texttt{ := } e_2 : \texttt{unit}$ if $e_1 : t \texttt{ ref}$ and $e_2 : t$.

In the last rule we see the use of type `unit`, which contains exactly one value `()`. A return value of `unit` usually means that an expression is evaluated for effect (and not the value it returns, since that carries no information).

Evaluation. As usual, expressions are evaluated from left to right. Once all the arguments to an operator have been reduced to values, the corresponding operation is carried out.

$$
\begin{array}{rcll}
\langle s; \texttt{ref}\ e\rangle & \stackrel{1}{\Longrightarrow} & \langle s'; \texttt{ref}\ e'\rangle & \text{if} \quad \langle s; e\rangle \stackrel{1}{\Longrightarrow} \langle s'; e'\rangle \\
\langle s; \texttt{ref}\ v\rangle & \stackrel{1}{\Longrightarrow} & \langle (s, c \mapsto v); c\rangle & \text{where } c \text{ is a new cell} \\
\\
\langle s; \texttt{!}e\rangle & \stackrel{1}{\Longrightarrow} & \langle s'; \texttt{!}e'\rangle & \text{if} \quad \langle s; e\rangle \stackrel{1}{\Longrightarrow} \langle s'; e'\rangle \\
\langle s; \texttt{!}c\rangle & \stackrel{1}{\Longrightarrow} & \langle s; v\rangle & \text{if } c \mapsto v \text{ in } s \\
\\
\langle s; e_1 \ \texttt{:=}\ e_2\rangle & \stackrel{1}{\Longrightarrow} & \langle s'; e_1' \ \texttt{:=}\ e_2\rangle & \text{if} \quad \langle s; e_1\rangle \stackrel{1}{\Longrightarrow} \langle s'; e_1'\rangle \\
\langle s; c \ \texttt{:=}\ e_2\rangle & \stackrel{1}{\Longrightarrow} & \langle s'; c \ \texttt{:=}\ e_2'\rangle & \text{if} \quad \langle s; e_2\rangle \stackrel{1}{\Longrightarrow} \langle s'; e_2'\rangle \\
\langle s; c \ \texttt{:=}\ v\rangle & \stackrel{1}{\Longrightarrow} & \langle s'; \texttt{()}\rangle & \text{where } s = s_1, c \mapsto v_0, s_2 \text{ and } s' = s_1, c \mapsto v, s_2
\end{array}
$$

The operational semantics above specifies exactly how expressions are evaluated. For example, in e_1 `:=` e_2 we first evaluate e_1 to a cell c (possibly affecting the store), then e_2 to a value v (possibly affecting the store further) and then modify the contents of c to contain v. This can lead to extremely obscure behavior (see example below) and one should avoid nesting expressions with effects. In the example we use the sequencing operator `(`e_1`;` e_2`)` which evaluates e_1 for its effect (discarding the value) and then returns the value of e_2. Note that this is different from the optional use of semi-colon to separate declarations.

```
val c = ref 5;           (* val c : int ref = ref(5) *)
val x = ((c := 3; c) := !(c := !c-1; c); !c); (* val x : int = 2 *)
```

Note any cell that might be referenced in a program must have a unique value. It must have a value, since an initial value must be supplied when a cell is created. The value must be unique since all cell labels in the store are different. Note also that there is no explicit operation to delete a cell from the store. Instead, a process called *garbage collection* will periodically remove cells which are no longer accessible. For example, after evaluation of the expression

```
let val c = ref true in !c end;
```

the cell created by the call to `ref true` is no longer accessible and it is safe to garbage collect it from the store.

4 Equality between Cells and Aliasing

Programming with mutable state is generally more difficult than pure programming, because it requires keeping track of the store in addition to the values of expressions. Changes to the store are not apparent in the type of functions, which means that explicit annotation of your code with invariants is even more important than for pure programs.

It is also possible to confuse a cell with its contents, but fortunately the type systems helps you in sorting them out. For the examples below we take advantage of the fact that cells permit equality. But note that c_1 `=` c_2 compares *cells*, not their contents.

The output from an ML top level is shown as a comment. First we create two cells (binding `c1` and `c2`), then increment the second. Now `c1` and `c2` are still different cells (`b1 = false`), but their contents is the same (`b2 = true`).

```
val c1 = ref 5;         (* val c1 : int ref = ref(5) *)
val c2 = ref 4;         (* val c2 : int ref = ref(4) *)
val _ = (c2 := !c2+1);  (*  *)
val b1 = (c1 = c2);     (* val b1 : bool = false *)
val b2 = (!c1 = !c2);   (* val b2 : bool = true *)
```

Next we can create an *alias* `c3` for the cell `c2`. Note that `c2` and `c3` are now bound to the *same* cell. Mutating this cell will thus affect `c2` and `c3`. Failure to keep track of different expressions denoting the same cell (a more general form of *aliasing*) is a common source of errors in programming with state.

```
val c3 = c2;            (* val c3 : int ref = ref(5) *)
val b3 = (c2 = c3);     (* val b3 : bool = true *)
val _ = (c3 := !c3+1);  (*  *)
c3;                     (* val it : int ref = ref(6) *)
c2;                     (* val it : int ref = ref(6) *)
```

5 Arrays

An array in ML is a fixed-size sequence of mutable cells. The signature `ARRAY` in the Standard ML Basis Library[1] provides some basic operations on arrays. Most of these are definable from operations analogous to those for references: create an array, mutate a cell in an array, and read the contents of a cell in an array, defined informally below.

```
exception Subscript
signature ARRAY =
sig
  eqtype 'a array
  val array : int * 'a -> 'a array
  val length : 'a array -> int
  val sub : 'a array * int -> 'a
  val update : 'a array * int * 'a -> unit
  ... more specifications ...
end
```

`exception Subscript` is a pervasive exception provided for arrays and similar structures.

t `array` is the type of arrays whose cells contain values of type *t*. Note that arrays admit equality, which is like cell equality (each call to `array` creates a new array).

`array` (*n*, *v*) creates a new array of size *n* where each cell is initialized with value *v*.

`length` (*a*) returns the size of the array *a*.

`sub` (*a*, *i*) returns the contents of cell *i* in array *a* if $0 \leq i <$ `length`(*a*) and raises exception `Subscript` otherwise.

`update` (*a*, *i*, *v*) mutates cell *i* in array *a* to contain the value *v* if $0 \leq i <$ `length`(*a*) and raises exception `Subscript` otherwise.

[1] presently accessible at `http://portal.research.bell-labs.com/orgs/ssr/sml/array.html` or through the course home page.