# CPSC 311:
# State **(DRAFT)**
# ("lec-state")

Joshua Dunfield

University of British Columbia

November 18, 2016

## 1 State

All of our Fun dialects have had only *immutable* bindings: During evaluation, once an identifier is bound to an expression, its meaning cannot change—it will have the same meaning as long as it is in scope (and not shadowed by another binding).

### 1.1 Classifying languages

Many languages have *mutable* state in some form:

- *By default, and idiomatic*: Fortran, Algol-60, Lisp, C, C++, Java, Smalltalk, ...
- *By default, but less idiomatic*: Racket
- *Not by default*: Standard ML, OCaml
- *By simulation*: Haskell

The line between "functional" and "imperative" is fuzzy, but I think most people would draw it somewhere around Racket. The line between "purely functional" and "impurely functional"—*purity* meaning a "lack of side effects (such as state)"—is usually drawn between ML and Haskell. That line is also subject to debate, however.

Starting from the top of the list, in languages like Java, most features are mutable by default (unless `const` is given).

Racket occupies a strange position in this space: fundamental binding operations like **define** and **let** are mutable, but "good Racket style" discourages you from exploiting this. A few language features in Racket, including lists, are genuinely immutable by default (Racket also has mutable lists, but not by default; as we saw in our discussion of classifying languages, different languages often provide the same behaviours and differ only in which behaviour is the default).

The ML languages are fairly consistent in being immutable by default. A value bound by a `let` in SML or OCaml cannot be mutated. Both languages do have features similar to Racket's boxes, but these must be used explicitly; the default is immutability. An exception that puts OCaml slightly nearer the top of the page: strings are mutable, as you can see from the following interaction with OCaml.

```
# let s = "abcd" ;;
val s : string = "abcd"
# String.set s 2 'r' ;;
- : unit = ()
# s ;;
- : string = "abrd"
```

(2016 update: I believe this has been changed in the latest version of OCaml.)

Haskell is usually considered "pure" or "purely functional", though there is debate about this too, partly because some people argue that nontermination is a side effect. In practice, Haskell has ample support (features like the appallingly named "monads") for an imperative style of programming. (As an aside, the techniques used to *implement* Haskell depend on mutable state!)

## 1.2 Defining state

The particular form of state we'll add to Fun is called *references* (using the ML terminology). A *reference* or *ref* is essentially a pointer to a *cell* (or "ref cell") whose contents can be mutated.

Modelling refs in a dynamic semantics requires significant changes. Now that we have an environment-based semantics, the environment $e\mathrm{n}v$ might seem to be a logical place to store the current contents of ref cells. That turns out to be a bad idea—we want the state of ref cells to survive a lexical scope, but not the state of binders (see the question below)—so instead we'll create a new animal, a *store* S. Like an environment, a store is basically a list:

| | | | |
|---|---|---|---|
| Stores | $S$ | $::=$ $\emptyset$ | empty store |
| | | $\mid \ell \triangleright v, S$ | location $\ell$ points to cell with contents $v$, followed by store $S$ |

Let's extend the concrete and abstract syntax.

| | | | |
|---|---|---|---|
| Expressions | ⟨E⟩ | ::= | ... |
| | | | \| {Ref ⟨E⟩} |
| | | | \| {Deref ⟨E⟩} |
| | | | \| {Setref ⟨E⟩ ⟨E⟩} |

The intended semantics is:

- {Ref ⟨E1⟩} evaluates ⟨E1⟩ and returns the location of a new cell (think of a location as a pointer);
- {Deref ⟨E1⟩} evaluates ⟨E1⟩, which must evaluate to a location, and returns the contents of the cell at that location;
- {Setref ⟨E1⟩ ⟨E2⟩} evaluates ⟨E1⟩, which must evaluate to a location $\ell$, then evaluates ⟨E2⟩ and puts that resulting value into the cell at location $\ell$.

The abstract syntax is extended correspondingly:

```
(define-type E
  ...
  [Ref (initial-contents E?)]
  [Deref (loc-expr E?)]
  [Setref (loc-expr E?) (new-contents E?)]
```

We aren't quite done, though: a `Ref` is how we *create* a new cell, not a *pointer* to a new cell. So we need one more variant:

```
(define-type E
  ...
  [Ref (initial-contents E?)]
  [Deref (loc-expr E?)]
  [Setref (loc-expr E?) (new-contents E?)]
  [Location (locsym symbol?)]
```

Racket has a built-in function called `gensym` that we'll use when we need a new location, to get a "fresh" symbol.

The point of a store is that its contents are mutable: evaluating an expression may change the store. So we need both an *input* store and an *output* store.

$\boxed{env; S \vdash e \Downarrow v; S'}$ Starting in environment $env$ and store $S$, evaluating $e$ produces value $v$ and updated store $S'$

$$\frac{env; S \vdash e \Downarrow v; S1}{env; S \vdash (\mathsf{Ref}\ e) \Downarrow (\mathsf{Location}\ \ell); \ell \triangleright v,\ S1}\ \text{??SEnv-ref}$$

What is $\ell$? It really doesn't matter, as long as it isn't already in S1. The judgment "$\ell$ fresh for S1" means that $\ell$ is not already mapped by S1.

$$\frac{\ell \text{ fresh for } S1 \qquad env; S \vdash e \Downarrow v; S1}{env; S \vdash (\mathsf{Ref}\ e) \Downarrow (\mathsf{Location}\ \ell); \ell \triangleright v,\ S1}\ \text{SEnv-ref}$$

When a new feature (like Ref) leads us to change the judgment form, we need to check two things:

- We can write rules for the new features.
- We can update the rules for old features.

We seem to have a rule for the new feature Ref, so we should try to update an old rule. We'll do the rule for Pair. We first need to update Eval-pair to use environments. Since pairs don't bind identifiers, this is straightforward:

$$\frac{env \vdash e1 \Downarrow v1 \qquad env \vdash e2 \Downarrow v2}{env \vdash (\mathsf{Pair}\ e1\ e2) \Downarrow (\mathsf{Pair}\ v1\ v2)}\ \text{Env-pair (stateless version)}$$

$$\frac{env; S \vdash e1 \Downarrow v1; S1 \qquad env; S1 \vdash e2 \Downarrow v2; S2}{env; S \vdash (\mathsf{Pair}\ e1\ e2) \Downarrow (\mathsf{Pair}\ v1\ v2); S2}\ \text{SEnv-pair}$$

This version of Env-pair says that, to evaluate a pair, we first evaluate $e1$ under the given store S, producing a (possibly) changed store S1; then, we evaluate $e2$ under S1, producing another store S2, which is the store produced by the entire evaluation of $(\mathsf{Pair}\ e1\ e2)$.

Following this pattern of passing stores along from premise to premise means that when we draw a derivation tree, and draw a line (really a curve) from the conclusion's starting store (to the

left of the turnstile $\vdash$) to the conclusion's result (to the right of the semicolon), the line looks like a thread. The store is "threaded through" the derivation tree.

Unlike previous evaluation rules for Pair, SEnv-pair specifies that an interpreter must evaluate the two expressions $e1$ and $e2$ in a particular order. The expression $e2$ *cannot* be evaluated before $e1$, because we need to evaluate $e1$ to know what S1 is.

(Adding the full, tedious set of error-handling rules to the old big-step semantics, or to the environment-based semantics without state, would also enforce this order. But now, it is enforced within the non-error rule. Small-step semantics also enforces this, using evaluation contexts $\mathcal{C}$.)

Before we try to update all the old rules for this different evaluation judgment, we should figure out how to evaluate the other new features:

$$\frac{\ell \text{ fresh for } S1 \qquad env; S \vdash e \Downarrow v; S1}{env; S \vdash (\mathsf{Ref}\ e) \Downarrow (\mathsf{Location}\ \ell); \ell \triangleright v,\ S1}\ \text{SEnv-ref}$$

$$\frac{env; S \vdash e \Downarrow (\mathsf{Location}\ \ell); S2 \qquad \textit{lookup-loc}(S2, \ell) = v}{env; S \vdash (\mathsf{Deref}\ e) \Downarrow v; S2}\ \text{SEnv-deref}$$

$$\frac{env; S \vdash e1 \Downarrow (\mathsf{Location}\ \ell); S1 \qquad env; S1 \vdash e2 \Downarrow v2; S2 \qquad \textit{update-loc}(S2, \ell, v2) = S3}{env; S \vdash (\mathsf{Setref}\ e1\ e2) \Downarrow v2; S3}\ \text{SEnv-setref}$$

The idea of *update-loc* is that $\textit{update-loc}(S2, \ell, v2) = S3$ where S3 is the same as S2, but with $\ell \triangleright \cdots$ replaced by $\ell \triangleright v2$. For example, if

$$S2 \quad = \quad \ell1 \triangleright (\mathsf{Num}\ 1), \ell2 \triangleright (\mathsf{Num}\ 0), \emptyset$$

then

$$\textit{update-loc}(S2, \ell2, (\mathsf{Num}\ 2)) \quad = \quad \ell1 \triangleright (\mathsf{Num}\ 1), \ell2 \triangleright (\mathsf{Num}\ 2), \emptyset$$

**Question:** Why do we need a separate store? What goes wrong if we use the environment to store ref cells?

Because then we would end up with a similar problem as not doing a freeness check during substitution: we could access an identifier that should be out of scope.

$$ePair \quad = \quad \big(\mathsf{Pair}\ (\mathsf{Let}\ x\ (\mathsf{Num}\ 1)\ (\mathsf{Id}\ x))\ (\mathsf{Let}\ y\ (\mathsf{Num}\ 2)\ (\mathsf{Id}\ x))\big)$$

Here, the second instance $(\mathsf{Id}\ x)$ is not in the scope of $(\mathsf{Let}\ x \ldots)$. But (if the environment contains ref cells), we have to thread the environment through; otherwise, the second component of the following pair $ePair'$ would be unable to see the effects of the first component. We expect $ePair'$ to evaluate to $(\mathsf{Pair}\ (\mathsf{Num}\ 1)\ (\mathsf{Num}\ 1))$ because the first component changes the contents of r from $(\mathsf{Num}\ 0)$ to $(\mathsf{Num}\ 1)$:

$$ePair' \quad = \quad \Big(\mathsf{Let}\ r\ (\mathsf{Ref}\ (\mathsf{Num}\ 0))\ \big(\mathsf{Pair}\ (\mathsf{Let}\ x\ (\mathsf{Num}\ 1)\ (\mathsf{Setref}\ r\ (\mathsf{Num}\ 1)))\ (\mathsf{Let}\ y\ (\mathsf{Num}\ 2)\ (\mathsf{Deref}\ r))\big)\Big)$$

If we thread the environment through, then in $ePair$, the binding of $(\mathsf{Id}\ x)$ would survive and could be used outside its scope; if we don't thread the environment through, $ePair'$ wouldn't behave as expected.

### 1.3 First implementation: `env-state.rkt`

We can **define-type** `Store` following the pattern of `Env`, and update `env-interp` to take *and* return a store. It's quite irritating, because Racket doesn't provide great support for returning pairs of things—I had to **define-type** `Config` to represent both an expression—the value $v$ being returned in $env; S1 \vdash e \Downarrow v; S2$—and the output store $S2$. But it can be done, and there are no big surprises.

Some of this could be done more easily in ML or Haskell, because those languages have more general "pattern matching" than **type-case**, so adjacent **type-case**s can be combined in one. We still have to look up locations in the store, and update a cell by constructing a new store with different contents for that one cell.

**Question:** Racket has boxes! Why not just use those, instead of going to all this trouble?

Good question. One answer is that we want to know that our interpreter follows the rules (is "sound with respect to the rules"). The code is annoying to read, but the correspondence to the rules is clear. If we use Racket's boxes as our locations, the code becomes much simpler, but we have to trust that Racket's semantics for boxes matches our rules.

I'm pretty sure it does match the rules, which is why I wrote another version of the interpreter that *does* use Racket boxes (`env-state-direct.rkt`—the word "direct" refers to using Racket's boxes directly).

Another answer is that, while we can (I think!) use Racket's boxes to correctly represent Fun's refs, we are depending on Racket's idea of a store being the same as ours. What if we wanted to allow "time travel" in Fun, where we could "checkpoint" an old store and "rewind" to it later? Racket—as far as I know—doesn't have that feature. So we'd need to either figure out how to checkpoint Racket boxes, or use the `env-state.rkt` representation where we don't use boxes.

This leads us to...

### 1.4 Second implementation: `env-state-direct.rkt`

In this interpreter, the operations on the `Store` **define-type** are simply calls to Racket's `unbox` and `set-box!`. Instead of reflecting the "threading" of the store in our interpreter, we assume that Racket's store behaves in that same way. (Again, I'm pretty sure it does.)

## 2 Translation dictionary

(Section added for lecture, 2016-11-14.)

| | Racket | Fun | C | C++ | Java |
|---|---|---|---|---|---|
| allocate | `(box e)` | (Ref $e$) | `malloc` | `new` | `new` |
| get | `(unbox e)` | (Deref $e$) | `*e` | `*e` | `e.fld` |
| set | `(set-box! e)` | (Setref $e1$ $e2$) | `*e1 = e2` | `*e1 = e2` | `e1.fld = e2` |

The Java column is approximate: a Java object has any number of instance variables, rather than just one.