


Extra problems, week 7

[Subscribe for email updates.](#)

 PINNED

 No tags yet. [+ Add Tag](#)

Charilaos Skiadas COMMUNITY TA · 24 days ago 

Kind of late for this week, but it's been hectic and I haven't had much time to work on notes (so apologies for no notes this week) and extra problems. But here is one extra problem for now. This picks up from the previous extra problem, and might not make too much sense on its own.

This week we will explore one more aspect of building an interpreter. We will show how to implement mutable state without using mutation in the meta-language (i.e. no refs). This will lead us to separate the "environment lookup" in two parts, and to create "store"s.

There are multiple approaches and I am not claiming mine will be better/special in some way. But it is an approach, intended to showcase some interesting features and ways of thinking.

Mutation without meta-mutation

We will start our investigation with attempting to implement the interpreter for a language that supports mutation, but without using mutation features of the meta-language (the language the interpreter is written in).

We will start with a definition of the language we will be looking at. The language will be more dynamic in nature, so we will not concern ourselves with any type-checking. It is however essentially the RSL language we've been looking at from last week:

```
datatype binop = ADD | SUB | MUL | DIV | MOD
datatype logop = EQ | LT | LE | NEQ
datatype rslVal =
  IntV of int
| BoolV of bool
| PairV of rslVal * rslVal
| RecordV of (string * rslVal) list
| LstV of rslVal list
| UnitV
| ClosV of { env: rslVal E.env, f: rslExp }
| RefV of location
and rslExp =
```

```

    Int of int
  | Bool of bool
  | Var of string
  | Unit
  | Binop of binop * rslExp * rslExp
  | Logop of logop * rslExp * rslExp
  | If of rslExp * rslExp * rslExp
  | Func of string option * string * rslExp
  | Call of rslExp * rslExp
  | Letstar of (string * rslExp) list * rslExp
  | Pair of rslExp * rslExp
  | Fst of rslExp
  | Snd of rslExp
  | Lst of rslExp list
  | Cons of rslExp * rslExp
  | Head of rslExp
  | Tail of rslExp
  | Null
  | IsNull of rslExp
  | Record of (string * rslExp) list
  | Field of string * rslExp
  | Ref of rslExp
  | Assign of rslExp * rslExp
  | ValOf of rslExp
  | Seq of rslExp list

```

Essentially, we have integers, booleans, variable bindings, pairs, records, lists, and functions. We no longer have a letrec, and in order to allow some limited recursion we have put the optional function name back into our function definitions. The main difference comes from the fact that our "variable" bindings no longer bind to values directly, but instead to "locations in memory holding the values", and we will discuss that in a moment. As far as the syntax is concerned:

1. We will need new value, `RefV`, that holds a "reference to a value".
2. We also need three new expressions: a `Ref` used to create new references, an `Assign`, and a `ValOf`. `Ref` will evaluate the expression, then create a "reference" to hold a location in memory containing that value, then returns that reference. `Assign` is given two expressions, and evaluates the two expressions in order. Then it checks that the result of the first expression is a reference, and if so stores the value computed by the second expression there. It returns unit as value. Lastly, `ValOf` retrieves the value stored in the reference.

Finally, in order to be able to do meaningful programs in this language, we need a way to chain expressions together, and that is what the new expression `Seq` does. It takes a list of expressions, evaluates all expressions in order, and returns the value of the last expression.

Now let us talk about how we will effect mutation "without meta-language mutation". So far here is how we have been looking at `eval-under-env`: Every expression is given an environment, and based on that environment it is meant to compute a value. It has however no other way to communicate

information to its caller. We need our expressions to somehow pass more information back to their callers. But what would that information be? They need to update the values of references stored in variables, so does this mean returning a new environment as well? Let us look at an example, in SML but translatable to RSL, to demonstrate some of the problems with that:

```
let val x = ref 3
    fun f () = !x
in let val y = x
    in y := 5; f ()
    end
end
```

We have a fundamental problem there. The call to our function `f` needs to be somehow informed about the fact that `x` no longer has the value 3. But we cannot do this via the environment, as `f`'s environment is determined by lexical scoping, while the actual value of `x` is determined by the dynamic state of the program at the time of the call.

So we need a new storage location, separate from the environment. That store will contain the values pointed to by references, and each expression evaluation may change this store in some way. But it does not change the environment: The environment now holds information about *where* the values are located, so it associates reference variables with *locations*, but it is the store that actually holds the values, and associates them with *locations*.

So let us start by defining this Store structure. It needs to define two types: A "location" type, and a "store" type to represent the entire collection of "locations":

```
signature STORE =
sig
  exception OutOfMemory
  type location
  type 'a store

  val empty: unit -> 'a store
  val new_location: 'a store * 'a -> location * 'a store
  val fetch: 'a store * location -> 'a
  val update: 'a store * (location * 'a) -> 'a store
  val free: 'a store * location -> 'a store
end
```

We will not use the last function, but it would be essential if we want to implement some sort of garbage collection mechanism. Notice that `new_location` returns a pair: Since we are trying to avoid mutation, we need to return the new store every time we try to update it.

You could try an advanced implementation of Store using red-black trees or some other structure, but for now here's the steps for a simple list-based implementation:

1. locations will be just integers.

2. `store` will be a pair of a list of `location * 'a` pairs and an integer counter, starting with an empty list and a counter of 0. The counter indicates the next available "location".
3. Creating a "new location" requires a value to store there. You would simply use the current counter to create a new value pair and return the counter, along with the updated store.
4. "Fetching" returns the value stored at a particular location (You can throw an Empty exception or something like that if somehow you end up with a location without stored value, which should not be happening).
5. "update" requires traversing the list till you find the location, then setting a new value in place of the old one.
6. For "free" simply remove the entry with that location.

With a store implementation under our belt, let us revisit our main structure for the language, which I will still refer to as RSL.

```
signature RSL =
sig
  exception EvalError of string
  structure E : ENV
  structure S : STORE
  type location
  sharing type S.location = location

  datatype binop = ADD | SUB | MUL | DIV | MOD
  datatype logop = EQ | LT | LE | NEQ
  datatype rslVal =
    IntV of int
  | BoolV of bool
  ...
  and rslExp =
    Int of int
  | Bool of bool
  ...

  val show: rslVal -> string
  val evalEnv: rslVal E.env -> rslVal S.store -> rslExp -> rslVal * rslVal S.store
  val eval: rslExp -> rslVal
end
```

The main difference you will notice is in the type signature for `evalEnv`. It now takes an environment and store (as well as the expression) as input, and it returns a pair of a value and an updated store. This is the crucial difference now, that every single evaluation needs to possibly update the store, and to return the updated store along with its result. Here is a skeleton for the solution:

```
structure Rsl :> RSL =
struct
  exception EvalError of string
```

```

structure E = Env
structure S = Store
type location = S.location

datatype ....
....

fun evalEnv env st e =
  case e of
    Int i => (IntV i, st)
  | ...
  | Binop (oper, e1, e2) =>
    let val (v1, st1) = evalEnv env st e1
        val (v2, st2) = evalEnv env st1 e2
    in (do_binop (oper, v1, v2), st2)
    end
  | ...

fun eval e = let val (v, st) = evalEnv E.empty S.empty e in v end
end

```

First note the kinds of values we return from `evalEnv`. For instance for a literal integer expression, we return the pair of the corresponding value and the unaltered state. A lot of the other cases will be similar. When it comes to evaluating multiple expressions however, care needs to be taken. Note that in `Binop` we evaluate the first value, and then use *the store it returned* as the store for the second value. This step is crucial now that we have the stores. While the environment is the same for both `e1`'s evaluation and `e2`'s evaluation, this is not true for the stores. Every operation updates the store, and we need to propagate that update further down the line. This forces a very sequential processing of the steps.





You should be able to translate most of the steps from the previous assignment to this one. There are 3 cases worthy of special mention:

1. In the `Ref e` case, you need to evaluate `e` to a value-store pair, and then use `new_location` on this new store to add the new reference. Make sure to return a `RefV` value along with the new store.
2. In the `Assign (e1, e2)` case, you need to evaluate the two terms in sequence just like for `Binop`. Then the first value must be a `RefV loc`, and then you need to use `update` for that location and the most up-to-date store.
3. In the `ValOf e1` case, you need to evaluate `e1`, resulting in a value and an updated store. The value must be a `RefV loc`, and you must "fetch" the value in that location from the *new store*.

These should be main changes needed! The hardest part would be writing tests to determine that your implementation is correct.

New post

To ensure a positive and productive discussion, please read our [forum posting policies](#) before posting.

B	<i>I</i>			 Link	<code>	 Pic	Math		Edit: Rich ▼	Preview
<div></div>										

☐ Make this post anonymous to other students

☒ Subscribe to this thread at the same time

Add post