EECS 662 Homework 2:
State and Typed References

J. Garrett Morris

September 24, 2017

Introduction

This homework will help you develop and demonstrate an understanding of *computational effects*, and in particular the interpretation of *state* in programming languages.

The homework distribution contains three Haskell files:

`Core.hs`	contains the interpreter for the core language (all your changes will go here)
`Main.hs`	contains scaffolding code to run both parsers on either standard input or files
`Sugar.hs`	contains parsing and desugaring functions for an extension of the core language with data types

The `Main.hs` file includes a driver program to simplify interaction with the interpreter. You can build the driver either using the included `Makefile`, or using the following GHC command:

```
ghc -o hw2 --make Main.hs
```

There are two modes of interacting with the interpreter. If you just call the interpreter (without passing any file names on the command line) then it functions as an interactive interpreter. You can write expressions, and those expression will be parsed, checked, and evaluated. For example:

```
$ ./hw2
> 2 + 2
(VInt 4,fromList [])
> 2 + True
ERROR: type error
> :q
$
```

Or, you can pass the interpreter a file (or series of files) to interpret. For example, if the file `Add.stlc` contains the text `2+2`, then:

```
$ ./hw2 Add.stlc
(VInt 4,fromList [])
$
```

The output includes both the computed value (`VInt 4`) and the memory at the end of evaluation. In these cases, the memory is empty, and so is displayed as an empty list. On the other hand, executing a simple program that uses the memory (after you have finished the problem set) should produce output like:

```
% ./hw2
> let k = ref 2 in put k (2 + get k); get k + 2
(VInt 6,fromList [(0,VInt 4)])
>
```

The homework distribution also includes several sample source files, `Incr.stlc`, `Tri.stlc`, and `Landin.stlc`. Each has its expected result as a comment on the first line of the file.

1 State

Your task in this problem is to implement *typed references*, building on the simple implementation of state we developed in class. The notion of state we developed in class had only a single state cell, which stored `Int` values. In this homework, you will extend that model to support an arbitrary number of state cells, each storing values of an arbitrary type. To do so:

- You will need to adapt the implementation of state from a single cell to a *map* from keys to values;
- You will need to implement typing rules for three new state-manipulating primitives; and,
- You will need to implement evaluation rules for the new primitives, in terms of the new implementation of state.

The typing task is independent of the other two. Evaluation depends on the new implementation of state.

> You *may* use any of the code developed in class, including `Lcst`n`.hs`, in developing your solutions to this problem. Keep in mind that the core language is more expressive than the language developed in class (for example, it contains sums), so you will not be able to transfer your solution directly from `Lcst`n`.hs`

1.1 Maps

We will model the state using a *map* from `Int` keys to `Value`s. This section introduces notation for maps and their Haskell implementations.

We will need three operations on map to describe the evaluation rules for language with state. First, we will need to know what value is associated with key k in map m; our notation for this will be $m(k)$. Second, we will need to update the value associated with key k in map m; our notation for this will be $m[k \mapsto v]$. Note that this notation describes a *new* map m', such that $m'(k) = v$ and $m'(j) = m(j)$ for all keys $j \neq k$. Finally, we will need to know the range of keys in a given map (so that we can make sure new keys do not overlap with any existing key). We will write $\mathrm{dom}(m)$ to denote the range of keys in m.

Haskell provides an implementation of maps, in the `Data.Map` library. *The `Core.hs` included in the homework distribution already imports the map library.* This library provides a type `Map` k v, which implements a map from keys of type k to values of type v. For our purposes, we will only be using maps of type `Map Int Value`. Each of the operations above is provided by one of the functions in the `Data.Map` library, as detailed on the table below. I have also listed several additional functions that you may find helpful in writing your implementation. Full documentation for the `Data.Map` library is available online (https://hackage.haskell.org/package/containers-0.5.10.2/docs/Data-Map.html).

Notation	Meaning	Haskell implementation
$\mathrm{dom}\, m$	Domain of m	`Map.keys m`
$m(k)$	Value associated with k in m	`Map.lookup k m`
$m[k \mapsto v]$	Updated version of m mapping k to v	`Map.insert k v m`
–	Empty map	`Map.empty`
–	Check whether map m is empty	`Map.null m`

1.2 Typed References

To support typed references, we need to add a new form of value vref k, denoting a reference into the map with key k, a new type `Ref` t to characterize references to values of type t, and three new expression forms `ref` e, `get` e, and `put` e_1 e_2, to create and manipulate typed references.

Notation	Meaning
`Ref` t	Type of references storing values of type t
vref k	Reference value, storing key k
`ref` e	Expression to generate a new reference with initial value e
`get` e	Expression to get the value associated with reference e
`put` e_1 e_2	Expression to put the value of e_2 into reference e_1

The following table reviews the notation for typing and evaluation.

Notation	Meaning
$\Gamma \vdash e : t$	Under assumptions Γ, expression e has type t
$e \mid m \Downarrow v \mid m'$	Expression e in state m evaluates to value v with final state m'
$[x \mapsto w]e \mid m \Downarrow v \mid m'$	With variable x mapped to value w, expression e in state m evaluates to value v with final state m'

The typing rules for typed references are as follows.

$$\frac{\Gamma \vdash e : t}{\Gamma \vdash \texttt{ref}\ e : \texttt{Ref}\ t} \qquad \frac{\Gamma \vdash e : \texttt{Ref}\ t}{\Gamma \vdash \texttt{get}\ e : t} \qquad \frac{\Gamma \vdash e_1 : \texttt{Ref}\ t \quad \Gamma \vdash e_2 : t}{\Gamma \vdash \texttt{put}\ e_1\ e_2 : 1}$$

The first rule states that, if e is a value of type t, then `ref` e is a value of type `Ref` t. The second rule says that, if e is a reference to a value of type t, then the result of retrieving that value is also of type t. The final rule says that to update a reference, the new value must match the type of value stored in the reference. (The expression `put` e_1 e_2 does not return any useful information; recall that 1 is the type of the unit value.)

Here are some examples of well-typed terms, with their types:

Term	Type
`ref` 4	`Ref Int`
`let` $z =$ `ref` 4 `in get` z	`Int`
`let` $z =$ `ref` 4 `in put` $z\ (4 +$ `get` $z)$	1

Here are some examples of ill-typed terms:

Term
`let` $z =$ `ref True in` $4 +$ `get` z
`let` $z =$ `ref True in put` $z\ 4$

The evaluation rules for typed references are as follows.

$$\frac{e \mid m \Downarrow v \mid m'}{\texttt{ref}\ e \mid m \Downarrow \mathsf{vref}\ k \mid m'[k \mapsto v]}\ (k \notin \mathrm{dom}(m'))$$

$$\frac{e \mid m \Downarrow \mathsf{vref}\ k \mid m'}{\texttt{get}\ e \mid m \Downarrow m'(k) \mid m'} \qquad \frac{e_1 \mid m \Downarrow \mathsf{vref}\ k \mid m_1 \quad e_2 \mid m_1 \Downarrow v \mid m_2}{\texttt{put}\ e_1\ e_2 \mid m \Downarrow \langle\rangle \mid m_2[k \mapsto v]}$$

The first rule covers new references, `ref` e. When evaluating `ref` e, two things happen. First, `ref` e returns a new reference $\mathsf{vref}\ k$. The side condition $k \notin \mathrm{dom}(m')$ requires that the key k not have already appeared in the map. Second, `ref` e updates the map to point key k to v, the result of evaluating e. The second rule covers retrieving values from references. Note that evaluating e to the reference $\mathsf{vref}\ k$ may itself have changed the state, so we have to look k up in m', not in m. The final rule handles updating references; $\langle\rangle$ is the *unit value* (i.e., the unique value of type 1).

Here are some examples of terms and the values resulting from their evaluation. *This table does not show the initial or final state.*

Term	Value
`let` $z =$ `ref` 2 `in get` $z + 2$	4
`let` $z =$ `ref` 2 `in put` $z\ ($`get` $z +$ `get` $z)$; `get` z	4
`let` $z =$ `ref` *False* `in put` $z\ ($`isz` $0)$; `if get` z `then` 1 `else` 0	1

As when we added computational effects in class, you will need to extend all the rules in the evaluator (i.e., the `eval` function) to take account of changes to the state, even if those terms do not manipulate the state directly. Here are two examples of how the existing evaluation rules are updated to take account of state. *This is not an exhaustive list; you will have to update the remaining rules following the same pattern.*

Old evaluation rule	New evaluation rule
$\dfrac{e_1 \Downarrow \lambda x.e \quad e2 \Downarrow v \quad [x \mapsto v]e \Downarrow w}{e_1\ e_2 \Downarrow w}$	$\dfrac{e_1 \mid m \Downarrow \lambda x.e \mid m_1 \quad e_2 \mid m_1 \Downarrow v \mid m_2 \quad [x \mapsto v]e \mid m_2 \Downarrow w \mid m_3}{e_1\ e_2 \mid m \Downarrow w \mid m_3}$
$\dfrac{e_1 \Downarrow v_1 \quad e_2 \Downarrow v_2}{e_1 + e_2 \Downarrow v_1 + v_2}$	$\dfrac{e_1 \mid m \Downarrow v_1 \mid m_1 \quad e_2 \mid m_1 \Downarrow v_2 \mid m_2}{e_1 + e_2 \mid m \Downarrow v_1 + v_2 \mid m_2}$

Running the interpreter. The provided `Core.hs` provides a function `run`, defined by

```
run r = r
```

This function is called from `Main.hs` to translate the result of `eval` into a printable value. When you update `eval` to return a state transforming function, you must also update `run` to provide an initial state value.

Submission Instructions

Your submission should include the three Haskell files from the original homework distribution, modified as necessary for your solution, the test files from the original distribution, and (at least) two new test cases, stored in separate files. Each of your test files should being with a comment line (i.e., a line beginning with two dashes) containing their expected result. Each of the provided tests follows this format; check there for details. Please submit these as a single file (`.zip` or `.tar`), via Blackboard.

Point allocation

Type checking	10
Evaluation	40
Test cases (2)	8
Total	58