Homework 10

Due date: Wednesday, December 2, 2015

The primary purpose of this assignment is to practice the use of monads. We reimplement our interpreter from Homework 9 using the state monad to chain the memory through the evaluation. We do not consider new language features in this version of our interpreter. In fact, we simplify the language by removing all parameter passing modes except pass-by-value.

Like last time, you will work on this assignment in pairs. However, note that each student needs to submit a write-up and are individually responsible for completing the assignment. You are welcome to talk about these exercises in larger groups. However, we ask that you write up your answers in pairs. Also, be sure to acknowledge those with which you discussed, including your partner and those outside of your pair.

Try to make your code as concise and clear as possible. Challenge yourself to find the most crisp, concise way of expressing the intended computation. This may mean using ways of expressing computation currently unfamilar to you.

Finally, make sure that your file compiles and runs (using Scala 2.11.7). A program that does not compile will *not* be graded.

Submission instructions. Upload to NYU classes exactly one file named as follows:

- `hw10-YourNYULogin.scala` with your answers to the coding exercises.

Replace `YourNYULogin` with your NYU login ID (e.g., I would submit `hw10-tw47.scala` and so forth). To help with managing the submissions, we ask that you rename your uploaded files in this manner.

Getting started. Download the code pack `hw10.zip` from the assignment section on the NYU classes page.

$$
\begin{array}{rcll}
n \in \mathit{Num} & & & \text{numbers (double)} \\
s \in \mathit{Str} & & & \text{strings} \\
a \in \mathit{Addr} & & & \text{addresses} \\
b \in \mathit{Bool} & ::= & \mathbf{true} \mid \mathbf{false} & \text{Booleans} \\
x \in \mathit{Var} & & & \text{variables} \\
f \in \mathit{Fld} & & & \text{field names} \\
\tau \in \mathit{Typ} & ::= & \mathbf{bool} \mid \mathbf{number} \mid \mathbf{string} \mid \mathbf{Undefined} \mid & \text{types} \\
& & (\overline{\cancel{mode}\ \tau}) \Rightarrow \tau_0 & \\
v \in \mathit{Val} & ::= & \mathbf{undefined} \mid n \mid b \mid s \mid a \mid & \text{values} \\
& & \mathbf{function}\ p\ (\overline{\cancel{mode}\ x : \tau})\ t\ e & \\
e \in \mathit{Expr} & ::= & x \mid v \mid \mathit{uop}\ e \mid e_1\ \mathit{bop}\ e_2 \mid e_1\ ?\ e_2\ :\ e_3 \mid & \text{expressions} \\
& & \mathbf{console.log}(e) \mid e_1\ (\overline{e}) \mid & \\
& & \mathit{mut}\ x = e_1 ; e_2 & \\
\mathit{uop} \in \mathit{Uop} & ::= & - \mid\ ! \mid \star & \text{unary operators} \\
\mathit{bop} \in \mathit{Bop} & ::= & + \mid - \mid \star \mid / \mid \texttt{===} \mid \texttt{!==} \mid < \mid > \mid & \text{binary operators} \\
& & \texttt{<=} \mid \texttt{>=} \mid \texttt{\&\&} \mid \texttt{||} \mid , \mid \texttt{|=} & \\
p & ::= & x \mid \epsilon & \text{function names} \\
t & ::= & :\tau \mid \epsilon & \text{return types} \\
\mathit{mut} \in \mathit{Mut} & ::= & \mathbf{const} \mid \mathbf{var} & \text{mutability} \\
M \in \mathit{Mem} & = & \mathit{Addr} \rightharpoonup \mathit{Val} & \text{memories}
\end{array}
$$

Figure 1: Abstract syntax of JakartaScript

Problem 1 JakartaScript Interpreter with State (20 Points)

We start from our language in Homework 9, but we remove all parameter passing modes except for call-by-value parameters. The syntax of the new language is shown in Figure 1. In Figure 2, we show the updated and new AST nodes.

Type Checking. The inference rules defining the typing relation are given in Figures 3 and 4. The only change compared to Homework 9 is that we no longer have to handle the different parameter passing modes. The new type inference function

```
def typeInfer(env: Map[String,(Mut,Typ)], e: Expr): Typ
```

has already been provided for you.

Evaluation. Your task is to implement a monadic version of the eval function in Homework 9.

The new big-step operational semantics is given in Figures 5 and 6. The rules are identical

```scala
sealed abstract class Expr extends Positional
...
/** Declarations */
case class Decl(mut: Mut, x: String, ed: Expr, eb: Expr) extends Expr
```
`Decl(`mut, x, e_d, e_b`)` $\quad mut\ x = e_d;\ e_b$

```scala
sealed abstract class Mut
case object MConst extends Mut
```
`MConst` **const**
```scala
case object MVar extends Mut
```
`MVar` **var**

```scala
/** Functions */
type Params = List[(String, Typ)]
case class Function(p: Option[String], xs: Params, t: Option[Typ], e: Expr) extends Val
```
`Function(`p`, List(`$\overline{(x,\tau)}$`),` t`,` e`)` **function** $p\,(\overline{x:\tau})\,t\ e$

```scala
/* Parameter Passing Modes */
sealed abstract class PMode
case object PConst extends PMode
```
`PConst` **const**
```scala
case object PName extends PMode
```
`PName` **name**
```scala
case object PVar extends PMode
```
`PVar` **var**
```scala
case object PRef extends PMode
```
`PRef` **ref**

```scala
/** Addresses and Mutation */
case object Assign extends Bop
```
`Assign` $=$
```scala
case object Deref extends Uop
```
`Deref` $*$
```scala
case class Addr private[ast] (addr: Int) extends Expr
```
`Addr(`a`)` a

```scala
/** Types */
sealed abstract class Typ
...
case class TFunction(ts: List[Typ], tret: Typ) extends Typ
```
`TFunction(List(`$\overline{\tau}$`),` τ_0`)` $(\overline{\tau}) \Rightarrow \tau_0$

Figure 2: Representing in Scala the abstract syntax of JAKARTASCRIPT. After each **`case class`** or **`case object`**, we show the correspondence between the representation and the concrete syntax.

$$\frac{}{\Gamma \vdash b : \mathbf{bool}}\ \textsc{TypeBool} \qquad \frac{}{\Gamma \vdash n : \mathbf{number}}\ \textsc{TypeNum} \qquad \frac{}{\Gamma \vdash s : \mathbf{string}}\ \textsc{TypeStr}$$

$$\frac{}{\Gamma \vdash \mathbf{undefined} : \mathbf{Undefined}}\ \textsc{TypeUndefined}$$

$$\frac{\Gamma \vdash e : \mathbf{number}}{\Gamma \vdash -e : \mathbf{number}}\ \textsc{TypeUMinus} \qquad \frac{\Gamma \vdash e : \mathbf{bool}}{\Gamma \vdash\ !\ e : \mathbf{bool}}\ \textsc{TypeNot}$$

$$\frac{\Gamma \vdash e_1 : \mathbf{bool} \quad \Gamma \vdash e_2 : \mathbf{bool} \quad \mathit{bop} \in \{\texttt{\&\&}, \texttt{||}\}}{\Gamma \vdash e_1\, \mathit{bop}\, e_2 : \mathbf{bool}}\ \textsc{TypeAndOr}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1\ \texttt{,}\ e_2 : \tau_2}\ \textsc{TypeSeq} \qquad \frac{\Gamma \vdash e : \tau}{\Gamma \vdash \mathbf{console.log}\,\texttt{(}e\texttt{)}\ : \mathbf{Undefined}}\ \textsc{TypePrint}$$

$$\frac{\Gamma \vdash e_1 : \mathbf{string} \quad \Gamma \vdash e_2 : \mathbf{string}}{\Gamma \vdash e_1 \texttt{+} e_2 : \mathbf{string}}\ \textsc{TypePlusStr}$$

$$\frac{\Gamma \vdash e_1 : \mathbf{number} \quad \Gamma \vdash e_2 : \mathbf{number} \quad \mathit{bop} \in \{\texttt{+}, \texttt{*}, \texttt{/}, \texttt{-}\}}{\Gamma \vdash e_1\, \mathit{bop}\, e_2 : \mathbf{number}}\ \textsc{TypeArith}$$

$$\frac{\Gamma \vdash e_1 : \tau \ \ \Gamma \vdash e_2 : \tau \ \ \tau \in \{\mathbf{number}, \mathbf{string}\} \ \ \mathit{bop} \in \{\texttt{>}, \texttt{>=}, \texttt{<}, \texttt{<=}\}}{\Gamma \vdash e_1\ \mathit{bop}\ e_2 : \mathbf{bool}}\ \textsc{TypeInequal}$$

$$\frac{\Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \tau \quad \tau \text{ has no function types} \quad \mathit{bop} \in \{\texttt{===}, \texttt{!==}\}}{\Gamma \vdash e_1\ \mathit{bop}\ e_2 : \mathbf{bool}}\ \textsc{TypeEqual}$$

$$\frac{\Gamma \vdash e_1 : \mathbf{bool} \quad \Gamma \vdash e_2 : \tau \quad \Gamma \vdash e_3 : \tau}{\Gamma \vdash e_1\ \texttt{?}\ e_2\ \texttt{:}\ e_3 : \tau}\ \textsc{TypeIf}$$

$$\frac{\Gamma \vdash e : \texttt{(}\tau_1\texttt{,}\ \ldots\texttt{,}\tau_n\texttt{)} \Rightarrow \tau \quad \Gamma \vdash e_1 : \tau_1 \quad \ldots \quad \Gamma \vdash e_n : \tau_n}{\Gamma \vdash e\,\texttt{(}e_1, \ldots, e_n\texttt{)} : \tau}\ \textsc{TypeCall}$$

Figure 3: Type checking rules for non-imperative primitives of JakartaScript (no changes compared to Homework 9)

$$\frac{\Gamma \vdash e_d : \tau_d \quad \Gamma' = \Gamma[x \mapsto (mut, \tau_d)] \quad \Gamma' \vdash e_b : \tau_b}{\Gamma \vdash mut\ x = e_d\texttt{;}\, e_b : \tau_b}\ \textsc{TypeDecl}$$

$$\frac{x \in \mathsf{dom}(\Gamma) \quad \Gamma(x) = (mut, \tau)}{\Gamma \vdash x : \tau}\ \textsc{TypeVar}$$

$$\frac{\Gamma(x) = (\mathbf{var}, \tau) \quad \Gamma \vdash e : \tau}{\Gamma \vdash x = e : \tau}\ \textsc{TypeAssignVar}$$

$$\frac{\begin{array}{c}\Gamma \vdash e : \texttt{(}\tau_1\texttt{,}\ \ldots\texttt{,}\tau_n\texttt{)}\ \Rightarrow \tau \\ \text{for all i:} \quad \Gamma \vdash e_i : \tau_i\end{array}}{\Gamma \vdash e\,\texttt{(}e_1, \ldots, e_n\texttt{)} : \tau}\ \textsc{TypeCall}$$

$$\frac{\begin{array}{l}\Gamma' = \Gamma[x_1 \mapsto (\mathbf{const}, \tau_1)] \ldots [x_n \mapsto (\mathbf{const}, \tau_n)] \\ \Gamma' \vdash e : \tau \quad \tau' = \texttt{(}\tau_1\texttt{,}\ \ldots\texttt{,}\tau_n\texttt{)}\ \Rightarrow \tau\end{array}}{\Gamma \vdash \mathbf{function}\,\texttt{(}x_1\texttt{:}\tau_1\texttt{,}\ \ldots\texttt{,}\ x_n\texttt{:}\tau_n\texttt{)}\, e : \tau'}\ \textsc{TypeFun}$$

$$\frac{\begin{array}{c}\Gamma' = \Gamma[x_1 \mapsto (\mathbf{const}, \tau_1)] \ldots [x_n \mapsto (\mathbf{const}, \tau_n)] \\ \Gamma' \vdash e : \tau \quad \tau' = \texttt{(}\tau_1\texttt{,}\ \ldots\texttt{,}\tau_n\texttt{)}\ \Rightarrow \tau\end{array}}{\Gamma \vdash \mathbf{function}\,\texttt{(}x_1\texttt{:}\tau_1\texttt{,}\ \ldots\texttt{,}\ x_n\texttt{:}\tau_n\texttt{)}\ \texttt{:}\tau\ e\ : \tau'}\ \textsc{TypeFunAnn}$$

$$\frac{\begin{array}{c}\Gamma' = \Gamma[x \mapsto \tau'][x_1 \mapsto (\mathbf{const}, \tau_1)] \ldots [x_n \mapsto (\mathbf{const}, \tau_n)] \\ \Gamma' \vdash e : \tau \quad \tau' = \texttt{(}\tau_1\texttt{,}\ \ldots\texttt{,}\tau_n\texttt{)}\ \Rightarrow \tau\end{array}}{\Gamma \vdash \mathbf{function}\ x\,\texttt{(}x_1\texttt{:}\tau_1\texttt{,}\ \ldots\texttt{,}x_n\texttt{:}\tau_n\texttt{)}\ \texttt{:}\tau\ e\ : \tau'}\ \textsc{TypeFunRec}$$

Figure 4: Type checking rules for imperative primitives of JakartaScript

to those given in Homework 9, except that we only support pass-by-value parameters in function expressions.

- The `eval` function now has the following signature

  ```
  def eval(e: Expr): State[Mem, Val]
  ```

  This function needs to be completed.

The `State[S, R]` type is defined for you and shown in Figure 7. The essence of `State[S, R]` is that it encapsulates a function of type `S => (S,R)`, which can be seen as a computation that returns a value of type `R` with an input-output state of type `S`. The `run` field holds precisely a function of the type `S=>(S,R)`. Seeing `State[Mem, Val]` as an encapsulated `Mem => (Mem,Val)`, we see how the judgment form $\langle M, e\rangle \Downarrow \langle M', v\rangle$ corresponds to the signature of `eval`.

For the evaluation rules that do not involve imperative features, the memory M is simply threaded through. The main advantage of using the encapsulated computation `State[Mem, Val]` is that this common-case threading is essentially put into the `State` data structure. One can view `State[Mem, Val]` as a "collection" that holds a computation over `Mem` that produces a `Val`, and thus, we can define a `map` method that creates an updated `State` "collection" holding the result of the callback `f` to the `map`. Applying `map` methods on different data structures is so frequent that Scala has an expression form

```
for (...) yield ...
```

that works for any data structure that defines a `map` method (cf., OSV).

We suggest that you extend the given template of the `eval` function case by case. For each case, first copy over your code for the corresponding case from the `eval` function of Homework 9. Then turn this code into a monadic version that hides the threading of the memory in the `State[Mem,Val]` data structure. Some of the cases are already provided for you.

$$\frac{}{\langle M, v\rangle \Downarrow \langle M, v\rangle}\ \textsc{EvalVal} \qquad \frac{\langle M, e\rangle \Downarrow \langle M', n\rangle}{\langle M, -\,e\rangle \Downarrow \langle M', -n\rangle}\ \textsc{EvalUMinus} \qquad \frac{\langle M, e\rangle \Downarrow \langle M', b\rangle}{\langle M, !\ e\rangle \Downarrow \langle M', !\ b\rangle}\ \textsc{EvalNot}$$

$$\frac{\langle M, e_1\rangle \Downarrow \langle M', \mathbf{true}\rangle \quad \langle M', e_2\rangle \Downarrow \langle M'', v_2\rangle}{\langle M, e_1\ \&\&\ e_2\rangle \Downarrow \langle M'', v_2\rangle}\ \textsc{EvalAndTrue}$$

$$\frac{\langle M, e_1\rangle \Downarrow \langle M', \mathbf{false}\rangle \quad \langle M', e_2\rangle \Downarrow \langle M'', v_2\rangle}{\langle M, e_1\ ||\ e_2\rangle \Downarrow \langle M'', v_2\rangle}\ \textsc{EvalOrFalse}$$

$$\frac{\langle M, e_1\rangle \Downarrow \langle M', \mathbf{false}\rangle}{\langle M, e_1\ \&\&\ e_2\rangle \Downarrow \langle M', \mathbf{false}\rangle}\ \textsc{EvalAndFalse} \qquad \frac{\langle M, e_1\rangle \Downarrow \langle M', \mathbf{true}\rangle}{\langle M, e_1\ ||\ e_2\rangle \Downarrow \langle M', \mathbf{true}\rangle}\ \textsc{EvalOrTrue}$$

$$\frac{\langle M, e_1\rangle \Downarrow \langle M', v_1\rangle \quad \langle M', e_2\rangle \Downarrow \langle M'', v_2\rangle}{\langle M, e_1\ \mathbf{,}\ e_2\rangle \Downarrow \langle M'', v_2\rangle}\ \textsc{EvalSeq}$$

$$\frac{\langle M, e\rangle \Downarrow \langle M', v\rangle \quad v \text{ printed}}{\langle M, \mathbf{console.log}\,(e)\,\rangle \Downarrow \langle M', \mathbf{undefined}\rangle}\ \textsc{EvalPrint}$$

$$\frac{\langle M, e_1\rangle \Downarrow \langle M', n_1\rangle \quad \langle M, e_2\rangle \Downarrow \langle M'', n_2\rangle \quad n = n_1 + n_2}{\langle M, e_1 + e_2\rangle \Downarrow \langle M'', n\rangle}\ \textsc{EvalPlusNum}$$

$$\frac{\langle M, e_1\rangle \Downarrow \langle M', s_1\rangle \quad \langle M', e_2\rangle \Downarrow \langle M'', s_2\rangle \quad s = s_1 + s_2}{\langle M, e_1 + e_2\rangle \Downarrow \langle M'', s\rangle}\ \textsc{EvalPlusStr}$$

$$\frac{\langle M, e_1\rangle \Downarrow \langle M', n_1\rangle \quad \langle M', e_2\rangle \Downarrow \langle M'', n_2\rangle \quad n = n_1\, bop\, n_2 \quad bop \in \{*, /, -\}}{\langle M, e_1\, bop\, e_2\rangle \Downarrow \langle M'', n\rangle}\ \textsc{EvalArith}$$

$$\frac{\langle M, e_d\rangle \Downarrow \langle M', v_d\rangle \quad \langle M', e_b[v_d/x]\rangle \Downarrow \langle M'', v_b\rangle}{\langle M, \mathbf{const}\ x = e_d\,\mathbf{;}\, e_b\rangle \Downarrow \langle M'', v_b\rangle}\ \textsc{EvalConstDecl}$$

$$\frac{\langle M, e_1\rangle \Downarrow \langle M', n_1\rangle \quad \langle M', e_2\rangle \Downarrow \langle M'', n_2\rangle \quad b = n_1\ bop\ n_2 \quad bop \in \{>, >=, <, <=\}}{\langle M, e_1\ bop\ e_2\rangle \Downarrow \langle M'', b\rangle}\ \textsc{EvalInequalNum}$$

$$\frac{\langle M, e_1\rangle \Downarrow \langle M', s_1\rangle \quad \langle M', e_2\rangle \Downarrow \langle M'', s_2\rangle \quad b = s_1\ bop\ s_2 \quad bop \in \{>, >=, <, <=\}}{\langle M, e_1\ bop\ e_2\rangle \Downarrow \langle M', b\rangle}\ \textsc{EvalInequalStr}$$

$$\frac{\langle M, e_1\rangle \Downarrow \langle M', v_1\rangle \quad \langle M', e_2\rangle \Downarrow \langle M'', v_2\rangle \quad b = (v_1\ bop\ v_2)}{\langle M, e_1\ bop\ e_2\rangle \Downarrow \langle M'', b\rangle}\ \textsc{EvalEqual}$$

$$\frac{\langle M, e_1\rangle \Downarrow \langle M', \mathbf{true}\rangle \quad \langle M', e_2\rangle \Downarrow \langle M'', v_2\rangle}{\langle M, e_1\ ?\ e_2\ :\ e_3\rangle \Downarrow \langle M'', v_2\rangle}\ \textsc{EvalIfThen}$$

$$\frac{\langle M, e_1\rangle \Downarrow \langle M', \mathbf{false}\rangle \quad \langle M', e_3\rangle \Downarrow \langle M'', v_3\rangle}{\langle M, e_1\ ?\ e_2\ :\ e_3\rangle \Downarrow \langle M'', v_3\rangle}\ \textsc{EvalIfElse}$$

Figure 5: Big-step operational semantics of non-imperative primitives of JakartaScript. The only change compared to Homework 8 is the threading of the memory.

$$\frac{\langle M, e\rangle \Downarrow \langle M', v\rangle \quad a \in \mathsf{dom}(M')}{\langle M, \star\, a = e\rangle \Downarrow \langle M'[a \mapsto v], v\rangle}\ \textsc{EvalAssignVar}$$

$$\frac{a \in \mathsf{dom}(M)}{\langle M, \star\, a\rangle \Downarrow \langle M, M(a)\rangle}\ \textsc{EvalDerefVar}$$

$$\frac{\begin{array}{c}\langle M, e_d\rangle \Downarrow \langle M_d, v_d\rangle \quad a \notin \mathsf{dom}(M_d) \\ M' = M_d[a \mapsto v_d] \quad \langle M', e_b[\star\, a/x]\rangle \Downarrow \langle M'', v_b\rangle\end{array}}{\langle M, \mathbf{var}\ x = v_d\,\mathtt{;}\ e_b\rangle \Downarrow \langle M'', v_b\rangle}\ \textsc{EvalVarDecl}$$

$$\frac{\begin{array}{c}\langle M, e_0\rangle \Downarrow \langle M', v_0\rangle \quad v_0 = \mathbf{function}\ x_0\,(\overline{x_i : \tau_i}) : \tau\ e \\ v_0' = (\mathbf{function}\,(\overline{x_i : \tau_i}) : \tau\ (e[v_0/x_0])) \quad \langle M', v_0'\,(\overline{e_i})\rangle \Downarrow \langle M'', v\rangle\end{array}}{\langle M, e_0\,(\overline{e_i})\rangle \Downarrow \langle M'', v\rangle}\ \textsc{EvalCallRec}$$

$$\frac{\begin{array}{c}\langle M, e_0\rangle \Downarrow \langle M', v_0\rangle \quad v_0 = \mathbf{function}\,(x_1 : \tau_1, \overline{x_i : \tau_i}) : \tau\ e \\ \langle M', e_1\rangle \Downarrow \langle M'', v_1\rangle \quad v_0' = (\mathbf{function}\,(\overline{x_i : \tau_i}) : \tau\ (e[v_1/x_1])) \\ \langle M'', v_0'\,(\overline{e_i})\rangle \Downarrow \langle M''', v\rangle\end{array}}{\langle M, e_0\,(e_1, \overline{e_i})\rangle \Downarrow \langle M''', v\rangle}\ \textsc{EvalCallConst}$$

$$\frac{\langle M, e_1\rangle \Downarrow \langle M', \mathbf{function}\,()\ t\ e\rangle \quad \langle M', e\rangle \Downarrow \langle M'', v\rangle}{\langle M, e_1\,()\rangle \Downarrow \langle M'', v\rangle}\ \textsc{EvalCall}$$

Figure 6: Big-step operational semantics of imperative primitives of JakartaScript.

```scala
sealed class State[S,R](run: S => (S,R)) {
  def apply(s: S) = run(s)

  def map[P](f: R => P): State[S,P] =
    new State((s: S) => {
      val (sp, r) = run(s)
      (sp, f(r))
    })

  def flatMap[P](f: R => State[S,P]): State[S,P] =
    new State((s: S) => {
      val (sp, r) = run(s)
      f(r)(sp) // same as f(r).apply(sp)
    })
}

object State {
  def apply[S,R](f: S => (S,R)): State[S,R] =
    new State(f)
  def init[S]: State[S,S] =
    State( s => (s, s) )
  def insert[S,R](r: R): State[S,R] =
    init map ( _ => r )
  def read[S,R](f: S => R) =
    init map ( s => (s, f(s)) )
  def write[S](f: S => S): State[S,Unit] =
    init flatMap ( s => State( _ => (f(s), ()) ) )
}
```

Figure 7: Implementation of the state monad