Homework 11

Due date: Wednesday, December 9, 2015

The primary purpose of this assignment is to explore objects in object-oriented languages. Concretely, we will extend JakartaScript with object literals, field dereference expressions, and field assignments. We extend our type checker to support the new features, but we do not yet add support for subtyping. At this point, we have many of the key features of JavaScript/TypeScript.

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 two files named as follows:

- `hw11-YourNYULogin.pdf` with your answers to the written questions (scanned, clearly legible handwritten write-ups are acceptable).
- `hw11-YourNYULogin.scala` with your answers to the coding exercises.

Replace `YourNYULogin` with your NYU login ID (e.g., I would submit `hw11-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 `hw11.zip` from the assignment section on the NYU classes page.

Problem 1 Objects and Subtyping (16 Points)

This is a warm-up exercise to get yourself familiar with objects (which you will implement in the second part of this homework) and subtyping (which you will implement in the upcoming final homework assignment).

(a) Indicate whether the following JakartaScript subtype relationships are true or false.

(i) `number <: number`

(ii) `bool <: number`

(iii) `{var f: number} <: {var f: any}`

(iv) `{var f: number} <: {const f: any}`

(v) `{const f: number} <: {var f: number}`

(vi) `{var f: number} <: {var g: number}`

(vii) `{const f: {}} <: {const f: any, var g: bool}`

(viii) `any => bool <: bool => any`

(ix) `bool => bool <: any => any`

(b) For each of the following programs: (1) say whether the program can be safely evaluated or not, i.e., whether the evaluation will get stuck or produce a value. If it can be safely evaluated, provide the computed value; (2) determine whether the program is well-typed with subtyping (i.e., with the rule TypeSub). If it is not, provide a brief explanation.

(i)

```
const x = {var f: 3};
const fun = function(y: {var f: number, const g: bool}) {
    y.f = 4; return y;
  };
fun(y).f
```

(ii)

```
const x = {var f: 3, const g: true};
const fun = function(y: {var f: number}) {
    return y;
  };
fun(y).f
```

(iii)

```
const x = {var f: 3, const g: true};
const fun = function(y: {var f: number}) {
    return y;
  };
fun(y).g
```

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

Figure 1: Abstract syntax of JakartaScript

(iv)

```
const x = {var f: 1, const g: true};
const y = {const f: false, var g: 2};
const z = true ? x : y;
z.f
```

Problem 2 JakartaScript Interpreter with Objects (24 Points)

We start from our language in Homework 10 and extend it with object literals and field dereference operations. The syntax of the new language is shown in Figure 1.

An object literal $\{\overline{\mathit{mut}\ f : e}\}$ is a sequence of field names associated with initialization expressions and mutabilites. The mutability of a field indicates whether the field can be reassigned a new value after the object literal has been evaluated. When an object literal is evaluated, we first evaluate the initialization expressions of the fields to obtain an object value $\{\overline{f : v}\}$. Note that object values are not actually values. In JavaScript, objects are dynamically allocated on the heap and then referenced with an extra level of indirection through a heap address. To model object allocation, object literals in JakartaScript evaluate to an address a, which is a value that references a memory location. This memory location that stores the object value obtained from the object literal.

Aliasing. The indirection introduced by dynamic allocation of objects means two program variables can reference the same object, which is called aliasing. With mutation, aliasing is now observable as demonstrated by the following example:

```
const x = { f : 1 };
const y = x;
x.f = 2;
console.log(y.f)
```

The code above should print `2` because `x` and `y` are aliases (i.e., reference to the same object). Aliasing makes programs more difficult to reason about and is often the source of subtle bugs.

In Figure 2, we show the updated and new AST nodes. Note that in our Scala representation of ASTs, we treat field dereference as a unary operator `FldDeref` that is parameterized by the dereferenced field name. That is, a field dereference expression $e\mathtt{.}f$ in JakartaScript is represented in Scala by the expression `UnOp(FldDeref(f), e)`. This design choice simplifies some aspects of our implementation. In particular, we will not have to extend the substitution function `subst` of our interpreter with an extra case for field dereference expressions.

Type Checking. The inference rules defining the typing relation are given in figures 3 and 4. The only change compared to Homework 10 are the new rules for typing objects, which are summarized in Figure 4. A template for the new type inference function

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

has been provided for you. The only missing cases are for the new rules in Figure 4. Your first task is to implement those missing cases.

Evaluation. The new big-step operational semantics is given in figures 5, 6, and 7. The rules are identical to those given in Homework 10, except that we add new rules for evaluating object literals, field dereference operations, and field assignments. These new rules are summarized in Figure 7.

Your second task is to update the `subst` and `eval` functions of the interpreter to account for the new language primitives.

- The `subst` function has the same signature as last time

  ```
  def subst(e: Expr, x: String, er: Expr): Expr
  ```

  The only missing case is the case for object literals. For object literals of the form $\mathtt{\{}\, mut_1\ f\mathtt{:}e_1, \ldots, mut_n\ f_n\mathtt{:}e_n \,\mathtt{\}}$ the substitution function should simply recurse into the field initialization expressions. That is, we have

  $$\mathtt{\{}\, mut_1\ f\mathtt{:}e_1, \ldots, mut_n\ f_n\mathtt{:}e_n \,\mathtt{\}}[e_r/x] = \mathtt{\{}\, mut_1\ f\mathtt{:}e_1[e_r/x], \ldots, mut_n\ f_n\mathtt{:}e_n[e_r/x] \,\mathtt{\}}$$

```scala
sealed abstract class Expr extends Positional

/** memory contents */
sealed trait Con

/** values */
sealed abstract class Val extends Expr with Con
...

/** Objects */
type Fld = String
case class Obj(fes: Map[Fld, (Mut, Expr)]) extends Expr
```

`Obj(Map(`$\overline{(f,(mut,e))}$`))` $\{\overline{mut\ f\,\texttt{:}\,e}\}$

```scala
case class FldDeref(f: String) extends Uop
```

`UnOp(FldDeref(`f`),` e`)` $e\texttt{.}f$

```scala
/** Object values (to be stored in memory) */
case class ObjVal(fvs: Map[Fld, Val]) extends Con
```

`ObjVal(Map(`$\overline{f,v}$`))` $\{\overline{f\,\texttt{:}\,v}\}$

```scala
/** Types */
sealed abstract class Typ
...
case class TObj(fts: Map[Fld, (Mut, Typ)]) extends Typ
```

`TObj(Map(`$\overline{(f,(mut,\tau))}$`))` $\{\overline{mut\ f\,\texttt{:}\,\tau}\}$

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.

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

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

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

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

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

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

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

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

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

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

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

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

$$\frac{x \in \mathsf{dom}(\Gamma) \quad \Gamma(x) = (\mathit{mut}, \tau)}{\Gamma \vdash x : \tau}\ \textsc{TypeVar} \qquad \frac{\Gamma(x) = (\textbf{var}, \tau) \quad \Gamma \vdash e : \tau}{\Gamma \vdash x = e : \tau}\ \textsc{TypeAssignVar}$$

$$\frac{\begin{array}{c}\Gamma \vdash e : \texttt{(}\tau_1\texttt{,} \ldots \texttt{,}\tau_n\texttt{)} \Rightarrow \tau \\ \text{for all i:} \quad \Gamma \vdash e_i : \tau_i\end{array}}{\Gamma \vdash e\,\texttt{(}e_1, \ldots, e_n\texttt{)} : \tau}\ \textsc{TypeCall} \qquad \frac{\begin{array}{c}\Gamma' = \Gamma[x_1 \mapsto (\textbf{const}, \tau_1)] \ldots [x_n \mapsto (\textbf{const}, \tau_n)] \\ \Gamma' \vdash e : \tau \quad \tau' = \texttt{(}\tau_1\texttt{,} \ldots \texttt{,}\tau_n\texttt{)} \Rightarrow \tau\end{array}}{\Gamma \vdash \textbf{function}\,\texttt{(}x_1 \texttt{:} \tau_1\texttt{,} \ldots\texttt{,}\ x_n \texttt{:} \tau_n\texttt{)}\ e : \tau'}\ \textsc{TypeFun}$$

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

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

Figure 3: Type checking rules for non-object primitives of JakartaScript (no changes compared to Homework 10)

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \ldots \quad \Gamma \vdash e_n : \tau_n}{\Gamma \vdash \{\, mut_1\ f_1 \mathbin{:} e_1 \texttt{,}\ \ldots \texttt{,}\ mut_n\ f_n \mathbin{:} e_n \,\} : \{\, mut_1\ f_1 \mathbin{:} \tau_1 \texttt{,}\ \ldots \texttt{,}\ mut_n\ f_n \mathbin{:} \tau_n \,\}}\ \textsc{TypeObj}$$

$$\frac{\begin{array}{c}\Gamma \vdash e : \{\, mut_1\ f_1 \mathbin{:} \tau_1 \texttt{,}\ \ldots \texttt{,}\ mut_n\ f_n \mathbin{:} \tau_n \,\} \\ f = f_i \quad \tau = \tau_i \quad i \in [1, n]\end{array}}{\Gamma \vdash e \texttt{.} f : \tau}\ \textsc{TypeDerefFld}$$

$$\frac{\begin{array}{c}\Gamma \vdash e_1 : \{\, mut_1\ f_1 \mathbin{:} \tau_1 \texttt{,}\ \ldots \texttt{,}\ mut_n\ f_n \mathbin{:} \tau_n \,\} \\ f = f_i \quad \tau = \tau_i \quad mut_i = \textbf{var} \quad i \in [1, n] \\ \Gamma \vdash e_2 : \tau\end{array}}{\Gamma \vdash e_1 \texttt{.} f = e_2 : \tau}\ \textsc{TypeAssignFld}$$

Figure 4: Type checking rules for objects of JakartaScript

Hint: use the `mapValues` method of the `Map` data structure in the Scala standard library to implement the recursive substitution inside the field initialization expressions in `fes`.

- The `eval` function has the same signature as last time

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

  That is, `eval` takes an expression and returns a state monad `State[Mem, Val]` that encapsulates a computation from an input memory state to an output memory state and a result value. We suggest to proceed as follows:

  (a) Copy over your solution from Homework 10 for the missing non-object cases (or see the sample solution for Homework 10 once it is released).

  (b) Then implement the new cases for the rules EvalDerefFld and EvalAssignFld. **Hint:** The helper functions `eToAddr` and `readObj` will be useful here.

  (c) Finally, implement the case for the rule EvalObj. The template that has been provided for this case splits the evaluation into two parts: (1) the construction of the map `fvs` for the object value to be stored in memory, (2) the actual allocation and update of the memory. Part (1) should be implemented with a `foldLeft` over the map `fes` holding the fields and associated expressions of the object literal. This `foldLeft` constructs a state monad `state` whose result is the map `fvs` for the object value. Part (2) should then be implemented with a `flatMap` call on the state monad returned by `foldLeft`. **Hint:** in part (1) use a **`for`** expression in the body of the function to which `foldLeft` is applied in order to extract the current field/value map from the accumulated state monad `state`. Then extend this map in the **`yield`** part with the current field `fi` and the value obtained from `fi`'s initialization expression `ei`.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Figure 5: Big-step operational semantics of non-imperative non-object primitives of JakartaScript (no changes compared to Homework 10).

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

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

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

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

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

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

Figure 6: Big-step operational semantics of imperative non-object primitives of JakartaScript (no changes compared to Homework 10).

$$\frac{\begin{array}{c}M_0 = M \quad \text{for all } i \in [1, n] : \langle M_{i-1}, e_i\rangle \Downarrow \langle M_i, v_i\rangle\\ a \notin \mathsf{dom}(M_n) \quad M' = M_n[a \mapsto \{\, f_1 : v_1,\ \ldots,\ f_n : v_n \,\}]\end{array}}{\langle M, \{\, mut_1\ f_1 : e_1,\ \ldots,\ mut_n\ f_n : e_n \,\}\rangle \Downarrow \langle M', a\rangle}\ \textsc{EvalObj}$$

$$\frac{\langle M, e\rangle \Downarrow \langle M', a\rangle \quad M(a) = \{\, \ldots, f : v, \ldots \,\}}{\langle M, e\,.\,f\rangle \Downarrow \langle M, v\rangle}\ \textsc{EvalDerefFld}$$

$$\frac{\begin{array}{c}\langle M, e_1\rangle \Downarrow \langle M', a\rangle \quad \langle M', e_2\rangle \Downarrow \langle M'', v_2\rangle\\ M''(a) = \{\, \ldots, f : v, \ldots \,\} \quad M''' = M''[a \mapsto \{\, \ldots, f : v_2, \ldots \,\}]\end{array}}{\langle M, e_1\,.\,f = e_2\rangle \Downarrow \langle M''', v_2\rangle}\ \textsc{EvalAssignFld}$$

Figure 7: Big-step operational semantics of objects in JakartaScript.