Homework 9

The primary purpose of this assignment is to explore mutation and imperative updates in programming languages. We explore this language feature together with the related notion of parameter passing modes. Concretely, we extend JakartaScript with mutable variables and parameter passing modes. Parameters are always passed by value in JavaScript, so the parameter passing modes in JakartaScript are an extension beyond JavaScript to illustrate a language design decision. We will update our type checker and interpreter from Homework 8 and see that mutation forces us to do a rather global refactoring of our interpreter.

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.

Problem 1 Parameter Passing Modes (12 Points)

This problem is meant as a warm-up exercise for Problem 2 where you will implement a version of JakartaScript that supports different parameter passing modes.

Consider the typed variant of JakartaScript that supports the different parameter passing modes we discussed. For each of the following programs, say whether the program is well-typed. If not, provide a brief explanation of the type error. If yes, compute the value that the program evaluates to and provide a brief explanation why you obtain that specific value.

(a)

```
1 const y = 3;
2 const f = function(let x: Num) {
3     x = x + 1; return x + y;
4   };
5 f(y) + y
```

(b)

```
1 let x = 3;
2 const f = function(name x: Num) {
3     return x + x;
4   };
5 f(x = x + 1) + x
```

(c)

```
1 let y = 3;
2 const f = function(let x: Num) {
3     x = x + 1; return x + y;
4   };
5 f(y * 2) + y
```

(d)

```
let x = 3;
const f = function(ref y: Num) {
    y = y + 1; return y + x;
  };
f(x) + x
```

(e)

```
let y = 3;
const f = function(ref x: Num) {
    x = x + 1; return x + y;
  };
f(y * 2) + y
```

Problem 2 JakartaScript Interpreter with State (28 Points)

At this point, we are used to extending our interpreter implementation by updating our type checker `typeInfer` and our interpreter `eval`. The syntax with the new extensions highlighted is shown in Figure 1.

Mutation. In this assignment, we add mutable variables declared as follows:

$$\textbf{let}\ x = e_d; e_b$$

and then include an assignment expression:

$$e_1 = e_2$$

that writes the value of e_2 to a location named by expression e_1.

Parameter Passing Modes. In this assignment, we can annotate function parameters with **const**, **let**, **name**, or **ref** to specify a parameter passing mode. The annotation **let** says that the parameter should be pass-by-value with an allocation for a new mutable parameter variable initialized to the argument value. The **name** and **ref** annotations specify pass-by-name and pass-by-reference, respectively. In Homework 8, all parameters were pass-by-value with an immutable variable. This mode is now captured by a **const** annotation. These "pass-by" terms are defined by their respective EvalCall rules in Figure 6.

Addresses a and dereference operations `*` a are included in program expressions e because they arise during evaluation. However, there is no way to explicitly write these expressions in the source program. Addresses and dereference expressions are examples of enrichments of program expressions as an intermediate form solely for evaluation.

In Figure 2, we show the updated and new AST nodes. Note that `Deref` is a `Uop` and `Assign` a `Bop`.

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

Figure 1: Abstract syntax of JakartaScript

Type Checking. The inference rules defining the typing relation are given in Figures 3 and 4. Similar to before, we implement type inference with the function

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

that you need to complete. Note that the type environment maps a variable name to a pair of a mutability (either `MConst` or `MLet`) and a type. A template for the `Function` case for `typeInfer` is provided that you may use if you wish.

In the implementation of our type checker we now distinguish two different kinds of type errors. Just like in Homework 8, we have regular static type errors for those situations where we detect a type mismatch. The second kind of type error is used to signal the cases where we expect an assignable expression (i.e., a mutable variable) but find a non-assignable expression. To signal this second kind of type error, we will use a Scala exception

```
case class LocTypeError(e: Expr) extends JsException
```

where `e` is the non-assignable expression. We also provide a helper function `locerr` to simplify throwing this exception.

```
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)   mut x = e_d; e_b

sealed abstract class Mut
case object MConst extends Mut
MConst const
case object MLet extends Mut
MLet let

/** Functions */
type Params = List[(String,(PMode, Typ))]
case class Function(p: Option[String], xs: Params, t: Option[Typ], e: Expr) extends Val
Function(p, List((x, (mode, τ))), t, e)  function p(mode x : τ)t e

/* Parameter Passing Modes */
sealed abstract class PMode
case object PConst extends PMode
PConst const
case object PName extends PMode
PName name
case object PLet extends PMode
PLet let
case object PRef extends PMode
PRef ref

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

/** Types */
sealed abstract class Typ
...
case class TFunction(ts: List[(PMode, Typ)], tret: Typ) extends Typ
TFunction(List((mode, τ)), τ_0)  (mode τ) ⇒ τ_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.

$$\frac{}{\Gamma \vdash b : \texttt{Bool}}\ \textsc{TypeBool} \qquad \frac{}{\Gamma \vdash n : \texttt{Num}}\ \textsc{TypeNum} \qquad \frac{}{\Gamma \vdash s : \texttt{String}}\ \textsc{TypeStr}$$

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

$$\frac{\Gamma \vdash e : \texttt{Num}}{\Gamma \vdash \texttt{-}\, e : \texttt{Num}}\ \textsc{TypeUMinus} \qquad \frac{\Gamma \vdash e : \texttt{Bool}}{\Gamma \vdash \texttt{!}\, e : \texttt{Bool}}\ \textsc{TypeNot}$$

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

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

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

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

$$\frac{\Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \tau \quad \tau \in \{\texttt{Num}, \texttt{String}\} \quad bop \in \{\texttt{>}, \texttt{>=}, \texttt{<}, \texttt{<=}\}}{\Gamma \vdash e_1\ bop\ e_2 : \texttt{Bool}}\ \textsc{TypeInequal}$$

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

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

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

Figure 3: Type checking rules for non-imperative primitives of JakartaScript (no changes compared to Homework 8)

For example, in the following program we declare a **`const`** variable `x`, which we then try to reassign:

```
1 const x = 3;
2 x = 2
```

The function `typeInfer` should reject this program throwing a `LocTypeError` with the expression `x` in the left side of the assignment on line 2 as argument.

Evaluation. We also update `subst` and `eval` from Homework 8. The required modifications to `subst` are minimal and have already been provided for you.

A big-step operational semantics is given in Figures 5 and 6. The big-step judgment form is now as follows:

$$\langle M, e\rangle \Downarrow \langle M', v\rangle$$

that says informally, "In input memory M, expression e evaluates to the output memory

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

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

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

$$\frac{\begin{array}{c}\Gamma \vdash e : (mode_1\ \tau_1, \ldots, mode_n\ \tau_n) \Rightarrow \tau \\ \text{for all i: } (mode_i \neq \textbf{ref} \text{ or } e_i \in \mathit{Var} \text{ and } \Gamma(e_i) = (\textbf{let}, \tau_i)) \quad \Gamma \vdash e_i : \tau_i\end{array}}{\Gamma \vdash e(e_1, \ldots, e_n) : \tau}\ \textsc{TypeCall}$$

$$\frac{\begin{array}{c}\Gamma' = \Gamma[x_1 \mapsto (mut(mode_1), \tau_1)] \ldots [x_n \mapsto (mut(mode_n), \tau_n)] \\ \Gamma' \vdash e : \tau \quad \tau' = (mode_1\ \tau_1, \ldots, mode_n\ \tau_n) \Rightarrow \tau\end{array}}{\Gamma \vdash \textbf{function}(mode_1\ x_1 : \tau_1, \ldots, mode_n\ x_n : \tau_n) e : \tau'}\ \textsc{TypeFun}$$

$$\frac{\begin{array}{c}\Gamma' = \Gamma[x_1 \mapsto (mut(mode_1), \tau_1)] \ldots [x_n \mapsto (mut(mode_n), \tau_n)] \\ \Gamma' \vdash e : \tau \quad \tau' = (mode_1\ \tau_1, \ldots, mode_n\ \tau_n) \Rightarrow \tau\end{array}}{\Gamma \vdash \textbf{function}(mode_1\ x_1 : \tau_1, \ldots, mode_n\ x_n : \tau_n) : \tau\ e\ : \tau'}\ \textsc{TypeFunAnn}$$

$$\frac{\begin{array}{c}\Gamma' = \Gamma[x \mapsto \tau'][x_1 \mapsto (mut(mode_1), \tau_1)] \ldots [x_n \mapsto (mut(mode_n), \tau_n)] \\ \Gamma' \vdash e : \tau \quad \tau' = (mode_1\ \tau_1, \ldots, mode_n\ \tau_n) \Rightarrow \tau\end{array}}{\Gamma \vdash \textbf{function}\ x(mode_1\ x_1 : \tau_1, \ldots, mode_n\ x_n : \tau_n) : \tau\ e\ : \tau'}\ \textsc{TypeFunRec}$$

$$mut(\textbf{const}) = mut(\textbf{name}) = \textbf{const}$$

$$mut(\textbf{let}) = mut(\textbf{ref}) = \textbf{let}$$

Figure 4: Type checking rules for imperative primitives of JakartaScript

M' and value v." The memory M is a map from addresses a to values. The presence of a memory M that gets updated during evaluation is the hallmark of imperative computation.

- The `eval` function now has the following signature

  ```
  def eval(m: Mem, e: Expr): (Mem, Val)
  ```

  corresponding to the updated operational semantics. This function needs to be completed.

Note that the change in the judgment form necessitates updating all rules, even those that do not involve imperative features as in Figure 5. Some rules require allocating fresh addresses. For example, EvalLetDecl specifies allocating a new address a and extending the memory, mapping a to the value v_d of the defining expression e_d. The address a is stated to be fresh by the constraint that $a \notin \mathsf{dom}(M_d)$. Then M_d is updated to M' by storing v_d at address a in M_d. In the implementation, you call `md.alloc(vd)` to get a pair `(mp, a)` consisting of a fresh address `a` and an updated memory state `mp` with the location of address `a` initialized to value `vd` in `md`, thus achieving both of the above steps in just one step.

You might notice that in our operational semantics, the memory M only grows and never shrinks during the course of evaluation. Our interpreter only ever allocates memory and never deallocates! This choice is fine in a mathematical model and for this assignment, but a production run-time system must somehow enable collecting garbage–allocated memory locations that are no longer used by the running program. Collecting garbage may be done manually by the programmer (as in C and C++) or automatically by a conservative garbage collector (as in JavaScript, Scala, Java, C#, and Python).

You might also notice that we have a single memory instead of a *stack of activation records* for local variables and a *heap* for objects as discussed in Computer Systems Organization. Our interpreter instead simply allocates memory for local variables when they are encountered (e.g., EvalLetDecl). It never deallocates, even though we know that with local variables, those memory locations become inaccessible by the program once the body of a declaration, respectively, function has been evaluation. The key observation is that the traditional stack is not essential for local variables but rather is an optimization for automatic deallocation based on block statements, respectively, function call-and-return.

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

$$\frac{\langle M, e_1\rangle \Downarrow \langle M_1, \texttt{true}\rangle \quad \langle M_1, e_2\rangle \Downarrow \langle M_2, v_2\rangle}{\langle M, e_1\ \texttt{\&\&}\ e_2\rangle \Downarrow \langle M_2, v_2\rangle}\ \textsc{EvalAndTrue}$$

$$\frac{\langle M, e_1\rangle \Downarrow \langle M_1, \texttt{false}\rangle \quad \langle M_1, e_2\rangle \Downarrow \langle M_2, v_2\rangle}{\langle M, e_1\ \texttt{||}\ e_2\rangle \Downarrow \langle M_2, v_2\rangle}\ \textsc{EvalOrFalse}$$

$$\frac{\langle M, e_1\rangle \Downarrow \langle M_1, \texttt{false}\rangle}{\langle M, e_1\ \texttt{\&\&}\ e_2\rangle \Downarrow \langle M_1, \texttt{false}\rangle}\ \textsc{EvalAndFalse} \qquad \frac{\langle M, e_1\rangle \Downarrow \langle M_1, \texttt{true}\rangle}{\langle M, e_1\ \texttt{||}\ e_2\rangle \Downarrow \langle M_1, \texttt{true}\rangle}\ \textsc{EvalOrTrue}$$

$$\frac{\langle M, e_1\rangle \Downarrow \langle M_1, v_1\rangle \quad \langle M_1, e_2\rangle \Downarrow \langle M_2, v_2\rangle}{\langle M, e_1\ \texttt{,}\ e_2\rangle \Downarrow \langle M_2, v_2\rangle}\ \textsc{EvalSeq}$$

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

$$\frac{\langle M, e_1\rangle \Downarrow \langle M_1, n_1\rangle \quad \langle M_1, e_2\rangle \Downarrow \langle M_2, n_2\rangle \quad n = n_1 + n_2}{\langle M, e_1\ \texttt{+}\ e_2\rangle \Downarrow \langle M_2, n\rangle}\ \textsc{EvalPlusNum}$$

$$\frac{\langle M, e_1\rangle \Downarrow \langle M_1, s_1\rangle \quad \langle M_1, e_2\rangle \Downarrow \langle M_2, s_2\rangle \quad s = s_1 + s_2}{\langle M, e_1\ \texttt{+}\ e_2\rangle \Downarrow \langle M_2, s\rangle}\ \textsc{EvalPlusStr}$$

$$\frac{\langle M, e_1\rangle \Downarrow \langle M_1, n_1\rangle \quad \langle M_1, e_2\rangle \Downarrow \langle M_2, n_2\rangle \quad n = n_1\ bop\ n_2 \quad bop \in \{\texttt{*}, \texttt{/}, \texttt{-}\}}{\langle M, e_1\ bop\ e_2\rangle \Downarrow \langle M_2, n\rangle}\ \textsc{EvalArith}$$

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

$$\frac{\langle M, e_1\rangle \Downarrow \langle M_1, n_1\rangle \quad \langle M_1, e_2\rangle \Downarrow \langle M_2, n_2\rangle \quad b = n_1\ bop\ n_2 \quad bop \in \{\texttt{>}, \texttt{>=}, \texttt{<}, \texttt{<=}\}}{\langle M, e_1\ bop\ e_2\rangle \Downarrow \langle M_2, b\rangle}\ \textsc{EvalInequalNum}$$

$$\frac{\langle M, e_1\rangle \Downarrow \langle M_1, s_1\rangle \quad \langle M_1, e_2\rangle \Downarrow \langle M_2, s_2\rangle \quad b = s_1\ bop\ s_2 \quad bop \in \{\texttt{>}, \texttt{>=}, \texttt{<}, \texttt{<=}\}}{\langle M, e_1\ bop\ e_2\rangle \Downarrow \langle M_2, b\rangle}\ \textsc{EvalInequalStr}$$

$$\frac{\langle M, e_1\rangle \Downarrow \langle M_1, v_1\rangle \quad \langle M_1, e_2\rangle \Downarrow \langle M_2, v_2\rangle \quad b = (v_1\ bop\ v_2)}{\langle M, e_1\ bop\ e_2\rangle \Downarrow \langle M_2, b\rangle}\ \textsc{EvalEqual}$$

$$\frac{\langle M, e_1\rangle \Downarrow \langle M_1, \texttt{true}\rangle \quad \langle M_1, e_2\rangle \Downarrow \langle M_2, v_2\rangle}{\langle M, e_1\ \texttt{?}\ e_2\ \texttt{:}\ e_3\rangle \Downarrow \langle M_2, v_2\rangle}\ \textsc{EvalIfThen}$$

$$\frac{\langle M, e_1\rangle \Downarrow \langle M_1, \texttt{false}\rangle \quad \langle M_1, e_3\rangle \Downarrow \langle M_3, v_3\rangle}{\langle M, e_1\ \texttt{?}\ e_2\ \texttt{:}\ e_3\rangle \Downarrow \langle M_3, v_3\rangle}\ \textsc{EvalIfElse}$$

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.

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

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

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

$$\frac{\begin{array}{c}\langle M, e_0\rangle \Downarrow \langle M_0, v_0\rangle \quad v_0 = \mathbf{function}\ x_0(\overline{mode_i\ x_i : \tau_i}) : \tau\ e \\ v_0' = (\mathbf{function}(\overline{mode_i\ x_i : \tau_i}) : \tau\ (e[v_0/x_0])) \quad \langle M_0, v_0'(\overline{e_i})\rangle \Downarrow \langle M', v\rangle\end{array}}{\langle M, e_0(\overline{e_i})\rangle \Downarrow \langle M', v\rangle}\ \textsc{EvalCallRec}$$

$$\frac{\begin{array}{c}\langle M, e_0\rangle \Downarrow \langle M_0, v_0\rangle \quad v_0 = \mathbf{function}(\mathbf{const}\ x_1 : \tau_1, \overline{mode_i\ x_i : \tau_i}) : \tau\ e \\ \langle M_0, e_1\rangle \Downarrow \langle M_1, v_1\rangle \quad v_0' = (\mathbf{function}(\overline{mode_i\ x_i : \tau_i}) : \tau\ (e[v_1/x_1])) \\ \langle M_1, v_0'(\overline{e_i})\rangle \Downarrow \langle M', v\rangle\end{array}}{\langle M, e_0(e_1, \overline{e_i})\rangle \Downarrow \langle M', v\rangle}\ \textsc{EvalCallConst}$$

$$\frac{\begin{array}{c}\langle M, e_0\rangle \Downarrow \langle M_0, v_0\rangle \quad v_0 = \mathbf{function}(\mathbf{name}\ x_1 : \tau_1, \overline{mode_i\ x_i : \tau_i}) : \tau\ e \\ v_0' = (\mathbf{function}(\overline{mode_i\ x_i : \tau_i}) : \tau\ (e[e_1/x_1])) \\ \langle M_0, v_0'(\overline{e_i})\rangle \Downarrow \langle M', v\rangle\end{array}}{\langle M, e_0(e_1, \overline{e_i})\rangle \Downarrow \langle M', v\rangle}\ \textsc{EvalCallName}$$

$$\frac{\begin{array}{c}\langle M, e_0\rangle \Downarrow \langle M_0, v_0\rangle \quad v_0 = \mathbf{function}(\mathbf{let}\ x_1 : \tau_1, \overline{mode_i\ x_i : \tau_i}) : \tau\ e \\ \langle M_0, e_1\rangle \Downarrow \langle M_1, v_1\rangle \quad v_0' = (\mathbf{function}(\overline{mode_i\ x_i : \tau_i}) : \tau\ (e[\ast a/x_1])) \\ a \notin \mathsf{dom}(M_1) \quad \langle M_1[a \mapsto v_1], v_0'(\overline{e_i})\rangle \Downarrow \langle M', v\rangle\end{array}}{\langle M, e_0(e_1, \overline{e_i})\rangle \Downarrow \langle M', v\rangle}\ \textsc{EvalCallLet}$$

$$\frac{\begin{array}{c}\langle M, e_0\rangle \Downarrow \langle M_0, v_0\rangle \quad v_0 = \mathbf{function}(\mathbf{ref}\ x_1 : \tau_1, \overline{mode_i\ x_i : \tau_i}) : \tau\ e \\ v_0' = (\mathbf{function}(\overline{mode_i\ x_i : \tau_i}) : \tau\ (e[\ast a/x_1])) \\ \langle M_0, v_0'(\overline{e_i})\rangle \Downarrow \langle M', v\rangle\end{array}}{\langle M, e_0(\ast a, \overline{e_i})\rangle \Downarrow \langle M', v\rangle}\ \textsc{EvalCallRef}$$

$$\frac{\langle M, e_1\rangle \Downarrow \langle M_1, \mathbf{function}()t\ e\rangle \quad \langle M_1, e\rangle \Downarrow \langle M', v\rangle}{\langle M, e_1()\rangle \Downarrow \langle M', v\rangle}\ \textsc{EvalCall}$$

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