CSCI 3155: Lab Assignment 4

Fall 2014: Due Friday November 14 at 7 pm

The primary purpose of this lab to understand the interplay between type checking and evaluation. Concretely, we will extend JAVASCRIPTY with immutable objects and extend our small-step interpreter from Lab 3. Unlike all prior language constructs, object expressions do not have an *a priori* bound on the number of sub-expressions because an object can have any number of fields. To represent objects, we will use collections from the Scala library and thus will need to get used to working with the Scala collection API.

As with each lab you will work on this assignment with your lab partners (as assigned in Lab 3; talk with your TA if this needs to be changed.).

You are welcome to talk about these questions 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 expression computation currently unfamilar to you.

Finally, make sure that your file compiles and runs (using Scala 2.10.3). A program that does not compile will *not* be graded.

Submission Instructions. You need to complete the following four files, which you will both need to submit a secure hash and demonstrate during your interview:

- `Lab4.md` with your answers to the written questions.
- `Lab4.scala` with your answers to the coding exercises
- `Lab4Spec.scala` with any updates to your unit tests.
- `Lab4.jsy` with a challenging test case for your JAVASCRIPTY interpreter.

Getting Started. Download the code pack `lab4.tar.gz` from the lab 4 tab on Piazza.

A suggested way to get familiar with Scala is to do some small lessons with Scala Koans (`http://www.scalakoans.org/`). Useful ones for Lab 4 are AboutHigherOrderFunctions, AboutLists, AboutPartialFunctions, and AboutInfixPrefixAndPostfixOperators.

1. **Warm-Up: Collections**. To implement our interpreter for JAVASCRIPTY with objects, we will need to make use of collections from Scala's library. One of the most fundamental operations that one needs to perform with a collection is to iterate over the elements of the collection. Like many other languages with first-class functions (e.g., Python, ML), the Scala

library provides various iteration operations via *higher-order functions*. Higher-order functions are functions that take functions as parameters. The function parameters are often called *callbacks*, and for collections, they typically specify what the library client wants to do for each element.

In this question, we practice both writing such higher-order functions in a library and using them as a client.

(a) Implement a function

```
def compressRec[A](l: List[A]): List[A]
```

that eliminates consecutive duplicates of list elements. If a list contains repeated elements they should be replaced with a single copy of the element. The order of the elements should not be changed.

Example:

```
scala> compressRec(List(1, 2, 2, 3, 3, 3))
res0: List[Int] = List(1, 2, 3)
```

This test has been provided for you in the template.

For this exercise, implement the function by direct recursion (e.g., pattern match on `l` and call `compressRec` recursively). Do not call any `List` library methods.

This exercise is from Ninety-Nine Scala Problems:

`http://aperiodic.net/phil/scala/s-99/` .

Some sample solutions are given there, which you are welcome to view. However, it is strongly encouraged that you first attempt this exercise before looking there. The purpose of the exercise is to get some practice for the later part of this homework. Note that the solutions there do not satisfy the requirements here (as they use library functions). If at some point you feel like you need more practice with collections, the above page is a good resource.

(b) Re-implement the compress function from the previous part as `compressFold` using the `foldRight` method from the `List` library. The call to `foldRight` has been provided for you. Do not call `compressFold` recursively or any other `List` library methods.

(c) Implement a higher-order recursive function

```
def mapFirst[A](f: A => Option[A])(l: List[A]): List[A]
```

that finds the first element in `l` where `f` applied to it returns a `Some(`a`)` for some value a. It should replace that element with a and leave `l` the same everywhere else.

Example:

```
scala> mapFirst((i: Int) => if (i < 0) Some(-i) else None)(List(1,2,-3,4,-5))
res0: List[Int] = List(1, 2, 3, 4, -5)
```

(d) Consider again the binary search tree data structure from Lab 1:

```
sealed abstract class Tree {
  def insert(n: Int): Tree = this match {
    case Empty => Node(Empty, n, Empty)
    case Node(l, d, r) =>
      if (n < d) Node(l insert n, d, r) else Node(l, d, r insert n)
  }

  def foldLeft[A](z: A)(f: (A, Int) => A): A = {
    def loop(acc: A, t: Tree): A = t match {
      case Empty => throw new UnsupportedOperationException
      case Node(l, d, r) => throw new UnsupportedOperationException
    }
    loop(z, this)
  }
}
case object Empty extends Tree
case class Node(l: Tree, d: Int, r: Tree) extends Tree
```

Here, we have implement the binary search tree `insert` as a method of `Tree`. For this exercise, complete the higher-order method `foldLeft`. This method performs an in-order traversal of the input tree `this` calling the callback `f` to accumulate a result. Suppose the in-order traversal of the input tree yields the following sequence of data values: $d_1, d_2, \ldots, d_n$. Then, `foldLeft` yields

$$\texttt{f}(\cdots(\texttt{f}(\texttt{f}(\texttt{z}, d_1), d_2))\cdots), d_n) .$$

We have provided a test client `sum` that computes the sum of all of the data values in the tree using your `foldLeft` method.

(e) Implement a function

```
def strictlyOrdered(t: Tree): Boolean
```

as a client of your `foldLeft` method that checks that the data values of `t` as an in-order traversal are in strictly accending order (i.e., $d_1 < d_2 < \cdots < d_n$).

Example:

```
scala> strictlyOrdered(treeFromList(List(1,1,2)))
res0: Boolean = false
```

2. **JavaScripty Type Checker**

As we have seen in the prior labs, dealing conversions and checking for dynamic type errors complicate the interpreter implementation. Some languages restrict the possible programs that it will execute to ones that it can guarantee will not result in a dynamic type error. This restriction of programs is enforced with an analysis phase after parsing known as *type checking*. Such languages are called *strongly, statically-typed*. In this lab, we will implement a strongly, statically-typed version of JAVASCRIPTY. We will not permit any type conversions and will guarantee the absence of dynamic type errors.

In this lab, we extend JAVASCRIPTY with types, multi-parameter functions, and objects/records (see Figure 1). We have a language of types τ and annotate function parameters with

expressions	$e ::= x \mid n \mid b \mid \textbf{undefined} \mid uop\, e_1 \mid e_1\ bop\ e_2 \mid e_1\ ?\ e_2 : e_3$
	$\mid \textbf{const}\ x = e_1;\ e_2 \mid \textbf{console.log}(e_1)$
	$\mid str \mid \textbf{function}\ p(\overline{x:\tau})\, tann\ e_1 \mid e_0(\overline{e})$
	$\mid \{\, f_1 : e_1, \ldots, f_n : e_n \,\} \mid e_1.f$
values	$v ::= n \mid b \mid \textbf{undefined} \mid str \mid \textbf{function}\ p(\overline{x:\tau})\, tann\ e_1$
	$\mid \{\, f_1 : v_1, \ldots, f_n : v_n \,\}$
unary operators	$uop ::= \texttt{-} \mid \texttt{!}$
binary operators	$bop ::= \texttt{,} \mid \texttt{+} \mid \texttt{-} \mid \texttt{*} \mid \texttt{/} \mid \texttt{===} \mid \texttt{!==} \mid \texttt{<} \mid \texttt{<=} \mid \texttt{>} \mid \texttt{>=} \mid \texttt{\&\&} \mid \vert\vert$
types	$\tau ::= \textbf{number} \mid \textbf{bool} \mid \textbf{string} \mid \textbf{Undefined} \mid (\overline{x:\tau}) \Rightarrow \tau' \mid \{\, f_1 : \tau_1; \ldots; f_n : \tau_n \,\}$
variables	x
numbers (doubles)	n
booleans	$b ::= \textbf{true} \mid \textbf{false}$
strings	str
function names	$p ::= x \mid \varepsilon$
field names	f
type annotations	$tann ::= : \tau \mid \varepsilon$
type environments	$\Gamma ::= \cdot \mid \Gamma[x \mapsto \tau]$

Figure 1: Abstract Syntax of JAVASCRIPTY

types. Functions can now take any number of parameters. We write a sequence of things using either an overbar or dots (e.g., $\overline{e}$ or $e_1, \ldots, e_n$ for a sequence of expressions). An object literal

$$\{\, f_1 : e_1, \ldots, f_n : e_n \,\}$$

is a comma-separted sequence of field names with initialization expressions surrounded by braces. Objects in this lab are more like records or C-`structs` as opposed to JavaScript objects, as we do not have any form of dynamic dispatch. Our objects in this lab are also immutable. The field read expression $e_1.f$ evaluates e_1 to an object value and then looks up the field named f. An object value is a sequence of field names associated with values. The type language τ includes base types for numbers, booleans, strings, and **undefined**, as well as constructed types for functions $(\overline{x:\tau}) \Rightarrow \tau'$ and objects $\{\, f_1 : \tau_1; \ldots; f_n : \tau_n \,\}$.

As an aside, we have chosen syntax that is compatible with the TypeScript language that adds typing to JavaScript. TypeScript aims to be fully compatible with JavaScript, so it is not as strictly typed as JAVASCRIPTY in this lab.

In Figure 2, we show the updated and new AST nodes. We update `Function` and `Call` for multiple parameters/arguments. Object literals and field read expressions are represented by `Object` and `GetField`, respectively.

In this lab, we implement a type checker that is very similar to a big-step interpreter. Instead of computing the value of an expression by recursively computing the value of each subexpression, we infer the type of an expression, by recursively inferring the type of each subexpression. An expression is *well-typed* if we can infer a type for it.

Given its similarity to big-step evaluation, we can formalize a type inference algorithm in a

```
/* Functions */
case class Function(p: Option[String], params: List[(String,Typ)], tann: Option[Typ],
                    e1: Expr) extends Expr
```
$\texttt{Function}(p,\ \overline{(x,\tau)},\ tann,\ e_1)\quad \textbf{function}\ p(\overline{x:\tau})\,tann\, e_1$

```
case class Call(e1: Expr, args: List[Expr]) extends Expr
```
$\texttt{Call}(e_1,\ \overline{e})\quad e_1(\overline{e})$

```
/* Objects */
case class Obj(fields: Map[String, Expr]) extends Expr
```
$\texttt{Object}(\overline{f:e})\quad \{\overline{f:e}\}$

```
case class GetField(e1: Expr, f: String) extends Expr
```
$\texttt{GetField}(e_1,\ f)\quad e_1.f$

```
/* Types */
case class TFunction(params: List[(String,Typ)], tret: Typ) extends Typ
```
$\texttt{TFunction}(\overline{x:\tau},\ \tau')\quad (\overline{x:\tau}) \Rightarrow \tau'$

```
case class TObj(tfields: Map[String, Typ]) extends Typ
```
$\texttt{TObj}(\overline{f:\tau})\quad \{\overline{f:\tau}\}$

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

similar way. In Figure 3, we define the judgment form $\Gamma \vdash e : \tau$ which says informally, "In type environment Γ, expression e has type τ." We will implement a function

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

that corresponds directly to this judgment form. It takes as input a type environment `env` (Γ) and an expression `e` (e) returns a type `Typ` (τ). It is informative to compare these rules with the big-step operational semantics from Lab 3.

The TYPEEQUALITY is slightly informal in stating

$$\tau \text{ has no function types}\,.$$

We intend this statement to say that the structure of τ has no function types. The helper function `hasFunctionTyp` is intended to return **true** if a function type appears in the input and **false** if it does not, so this statement can be checked by taking the negation of a call to `hasFunctionTyp`.

To signal a type error, we will use a Scala exception

```
case class StaticTypeError(tbad: Typ, esub: Expr, e: Expr) extends Exception
```

where `tbad` is the type that is inferred sub-expression `esub` of input expression `e`. These arguments are used to construct a useful error message. We also provide a helper function `err` to simplify throwing this exception.

We suggest the following step-by-step order to complete `typeInfer`.

1. First, complete the cases for the basic expressions excluding `Function`, `Call`, `Obj`, and `GetField`.

$\boxed{\Gamma \vdash e : \tau}$

$$\textsc{TypeVar}\quad \frac{}{\Gamma \vdash x : \Gamma(x)}$$

$$\textsc{TypeNeg}\quad \frac{\Gamma \vdash e_1 : \mathbf{number}}{\Gamma \vdash -e_1 : \mathbf{number}}$$

$$\textsc{TypeNot}\quad \frac{\Gamma \vdash e_1 : \mathbf{bool}}{\Gamma \vdash\, !e_1 : \mathbf{bool}}$$

$$\textsc{TypeSeq}\quad \frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1\,,\,e_2 : \tau_2}$$

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

$$\textsc{TypePlusString}\quad \frac{\Gamma \vdash e_1 : \mathbf{string} \qquad \Gamma \vdash e_2 : \mathbf{string}}{\Gamma \vdash e_1 + e_2 : \mathbf{string}}$$

$$\textsc{TypeInequalityNumber}\quad \frac{\Gamma \vdash e_1 : \mathbf{number} \qquad \Gamma \vdash e_2 : \mathbf{number} \qquad \mathit{bop} \in \{<,<=,>,>=\}}{\Gamma \vdash e_1\ \mathit{bop}\ e_2 : \mathbf{bool}}$$

$$\textsc{TypeInequalityString}\quad \frac{\Gamma \vdash e_1 : \mathbf{string} \qquad \Gamma \vdash e_2 : \mathbf{string} \qquad \mathit{bop} \in \{<,<=,>,>=\}}{\Gamma \vdash e_1\ \mathit{bop}\ e_2 : \mathbf{bool}}$$

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

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

$$\textsc{TypePrint}\quad \frac{\Gamma \vdash e_1 : \tau_1}{\Gamma \vdash \mathbf{console.log}(e_1) : \mathbf{Undefined}}$$

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

$$\textsc{TypeConst}\quad \frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma[x \mapsto \tau_1] \vdash e_2 : \tau_2}{\Gamma \vdash \mathbf{const}\ x = e_1; e_2 : \tau_2}$$

$$\textsc{TypeCall}\quad \frac{\Gamma \vdash e : (x_1 : \tau_1, \ldots, x_n : \tau_n) \Rightarrow \tau \qquad \Gamma \vdash e_1 : \tau_1 \qquad \cdots \qquad \Gamma \vdash e_n : \tau_n}{\Gamma \vdash e(e_1, \ldots, e_n) : \tau}$$

$$\textsc{TypeObject}\quad \frac{\Gamma \vdash e_i : \tau_i \qquad (\text{for all } i)}{\Gamma \vdash \{\ldots, f_i : e_i, \ldots\} : \{\ldots, f_i : \tau_i, \ldots\}}$$

$$\textsc{TypeGetField}\quad \frac{\Gamma \vdash e : \{\ldots, f : \tau, \ldots\}}{\Gamma \vdash e.f : \tau}$$

$$\textsc{TypeNumber}\quad \frac{}{\Gamma \vdash n : \mathbf{number}}$$

$$\textsc{TypeBool}\quad \frac{}{\Gamma \vdash b : \mathbf{bool}}$$

$$\textsc{TypeString}\quad \frac{}{\Gamma \vdash \mathit{str} : \mathbf{string}}$$

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

$$\textsc{TypeFunction}\quad \frac{\Gamma[x_1 \mapsto \tau_1] \cdots [x_n \mapsto \tau_n] \vdash e : \tau \qquad \tau' = (x_1 : \tau_1, \ldots, x_n : \tau_n) \Rightarrow \tau}{\Gamma \vdash \quad \mathbf{function}\ (x_1 : \tau_1, \ldots, x_n : \tau_n)\ e \quad : \tau'}$$

$$\textsc{TypeFunctionAnn}\quad \frac{\Gamma[x_1 \mapsto \tau_1] \cdots [x_n \mapsto \tau_n] \vdash e : \tau \qquad \tau' = (x_1 : \tau_1, \ldots, x_n : \tau_n) \Rightarrow \tau}{\Gamma \vdash \quad \mathbf{function}\ (x_1 : \tau_1, \ldots, x_n : \tau_n) : \tau\ e \quad : \tau'}$$

$$\textsc{TypeRecFunction}\quad \frac{\Gamma[x \mapsto \tau'][x_1 \mapsto \tau_1] \cdots [x_n \mapsto \tau_n] \vdash e : \tau \qquad \tau' = (x_1 : \tau_1, \ldots, x_n : \tau_n) \Rightarrow \tau}{\Gamma \vdash \quad \mathbf{function}\ x(x_1 : \tau_1, \ldots, x_n : \tau_n) : \tau\ e \quad : \tau'}$$

Figure 3: Typing of JAVASCRIPTY.

2. Then, work on these remaining cases. These cases use collections, so be sure to complete the previous question before attempting these cases. You can also work on `step` before finishing `typeInfer`. **Hints**: Helpful library methods here include `map`, `foldLeft`, `zipped`, `foreach`, and `mapValues`. You may want to use `zipped` in the `Call` case to match up formal parameters and actual arguments.

3. **JavaScripty Small-Step Interpreter**

   In this question, we update `substitute` and `step` from Lab 3 for multi-parameter functions and objects. Because of type checking, the `step` cases can simplified greatly. We eliminate the need to perform conversions and should no longer throw `DynamicTypeError`.

   We introduce another Scala exception type

   case class `StuckError(e: Expr)` **extends** `Exception`

   that should be thrown when there is no possible next step. This exception looks a lot like `DynamicTypeError` except that the intent is that it should never be raised. It is intended to signal a coding error in our interpreter rather than an error in the JAVASCRIPTY test input.

   In particular, if the JAVASCRIPTY expression e passed into `step` is closed and well-typed (i.e., judgment $\cdot \vdash e : \tau$ holds meaning `inferType(`e`)` does not throw `StaticTypeError`), then `step` should never throw a `StuckError`. This property of JAVASCRIPTY is known as *type safety*.

   A small-step operational semantics is given in Figure 4. This semantics no longer has conversions compared to Lab 3. It is much simpler because of type checking (e.g., even with the addition of objects, it fits on one page).

   As specified, SEARCHOBJECT is non-deterministic. As we view objects as an unordered set of fields, it says an object expression takes can take a step by stepping on any of its component fields. To match the reference implementation, you should make the step go on the first non-value as given by the left-to-right iteration of the collection using `Map.find`.

   We suggest the following step-by-step order to complete `substitute` and `step`.

   1. First, complete the basic expression cases not given in the template (e.g., DOMINUS, DOIFTRUE).

   2. Then, work on the object cases. These are actually simpler than the function cases. **Hint**: Note that field names are different than variable names. Object expressions are not variable binding constructs–what does that mean about `substitute` for them?

   3. Then, work on the function cases. **Hints**: You might want to use your `mapFirst` function from the warm-up question. Helpful library methods here include `map`, `foldRight`, `zipped`, and `forall`,

$$\boxed{e \longrightarrow e'}$$

$$\textsc{DoNeg}\ \frac{n' = -n}{-n \longrightarrow n'} \qquad \textsc{DoNot}\ \frac{b' = \neg b}{!\,b \longrightarrow b'} \qquad \textsc{DoSeq}\ \frac{}{v_1\,,\, e_2 \longrightarrow e_2} \qquad \textsc{DoArith}\ \frac{n' = n_1 \mathit{\ bop\ } n_2 \qquad \mathit{bop} \in \{+,-,*,/\}}{n_1 \mathit{\ bop\ } n_2 \longrightarrow n'} \qquad \textsc{DoPlusString}\ \frac{str' = str_1 + str_2}{str_1 + str_2 \longrightarrow str'}$$

$$\textsc{DoInequalityNumber}\ \frac{b' = n_1 \mathit{\ bop\ } n_2 \qquad \mathit{bop} \in \{<,<=,>,>=\}}{n_1 \mathit{\ bop\ } n_2 \longrightarrow b'} \qquad \textsc{DoInequalityString}\ \frac{b' = str_1 \mathit{\ bop\ } str_2 \qquad \mathit{bop} \in \{<,<=,>,>=\}}{str_1 \mathit{\ bop\ } str_2 \longrightarrow b'}$$

$$\textsc{DoEquality}\ \frac{b' = (v_1 \mathit{\ bop\ } v_2) \qquad \mathit{bop} \in \{===,!==\}}{v_1 \mathit{\ bop\ } v_2 \longrightarrow b'} \qquad \textsc{DoAndTrue}\ \frac{}{\mathbf{true}\ \&\&\ e_2 \longrightarrow e_2} \qquad \textsc{DoAndFalse}\ \frac{}{\mathbf{false}\ \&\&\ e_2 \longrightarrow \mathbf{false}}$$

$$\textsc{DoOrTrue}\ \frac{}{\mathbf{true}\ ||\ e_2 \longrightarrow \mathbf{true}} \qquad \textsc{DoOrFalse}\ \frac{}{\mathbf{false}\ ||\ e_2 \longrightarrow e_2} \qquad \textsc{DoPrint}\ \frac{v_1 \text{ printed}}{\mathbf{console.log}(v_1) \longrightarrow \mathbf{undefined}} \qquad \textsc{DoIfTrue}\ \frac{}{\mathbf{true}\ ?\ e_2 : e_3 \longrightarrow e_2}$$

$$\textsc{DoIfFalse}\ \frac{}{\mathbf{false}\ ?\ e_2 : e_3 \longrightarrow e_3} \qquad \textsc{DoConst}\ \frac{}{\mathbf{const}\ x = v_1;\ e_2 \longrightarrow e_2[v_1/x]} \qquad \textsc{DoCall}\ \frac{v = \mathbf{function}\ (x_1 : \tau_1, \ldots, x_n : \tau_n)\, \mathit{tann}\ e}{v(v_1, \ldots v_n) \longrightarrow e[v_n/x_n] \cdots [v_1/x_1]}$$

$$\textsc{DoCallRec}\ \frac{v = \mathbf{function}\ x(x_1 : \tau_1, \ldots, x_n : \tau_n)\, \mathit{tann}\ e}{v(v_1, \ldots v_n) \longrightarrow e[v_n/x_n] \cdots [v_1/x_1][v/x]} \qquad \textsc{DoGetField}\ \frac{}{\{f_1 : v_1, \ldots, f_i : v_i, \ldots, f_n : v_n\}.f_i \longrightarrow v_i} \qquad \textsc{SearchUnary}\ \frac{e_1 \longrightarrow e_1'}{\mathit{uop}\, e_1 \longrightarrow \mathit{uop}\, e_1'}$$

$$\textsc{SearchBinary}_1\ \frac{e_1 \longrightarrow e_1'}{e_1 \mathit{\ bop\ } e_2 \longrightarrow e_1' \mathit{\ bop\ } e_2} \qquad \textsc{SearchBinary}_2\ \frac{e_2 \longrightarrow e_2'}{v_1 \mathit{\ bop\ } e_2 \longrightarrow v_1 \mathit{\ bop\ } e_2'} \qquad \textsc{SearchPrint}\ \frac{e_1 \longrightarrow e_1'}{\mathbf{console.log}(e_1) \longrightarrow \mathbf{console.log}(e_1')}$$

$$\textsc{SearchIf}\ \frac{e_1 \longrightarrow e_1'}{e_1\ ?\ e_2 : e_3 \longrightarrow e_1'\ ?\ e_2 : e_3} \qquad \textsc{SearchConst}\ \frac{e_1 \longrightarrow e_1'}{\mathbf{const}\ x = e_1;\ e_2 \longrightarrow \mathbf{const}\ x = e_1';\ e_2} \qquad \textsc{SearchCall}_1\ \frac{e \longrightarrow e'}{e(e_1, \ldots, e_n) \longrightarrow e'(e_1, \ldots, e_n)}$$

$$\textsc{SearchCall}_2\ \frac{e_i \longrightarrow e_i'}{\big(\mathbf{function}\ p(\overline{x : \tau})\ e\big)(v_1, \ldots, v_{i-1}, e_i, \ldots, e_n) \longrightarrow \big(\mathbf{function}\ p(\overline{x : \tau})\ e\big)(v_1, \ldots, v_{i-1}, e_i', \ldots, e_n)}$$

$$\textsc{SearchObject}\ \frac{e_i \longrightarrow e_i'}{\{\ldots, f_i : e_i, \ldots\} \longrightarrow \{\ldots, f_i : e_i', \ldots\}} \qquad \textsc{SearchGetField}\ \frac{e_1 \longrightarrow e_1'}{e_1.f \longrightarrow e_1'.f}$$

Figure 4: Small-step operational semantics of JAVASCRIPTY