CSCI 3155: Lab Assignment 5

Fall 2014: Due Friday December 5 at 7 pm

The primary purpose of this lab to explore mutation or imperative updates in programming languages. We also explore two related language considerations: parameter passing modes and casting. Concretely, we extend JAVASCRIPTY with mutable variables and objects, type declarations, limited type casting, and parameter passing modes. At this point, we have many of the key features of JavaScript/TypeScript, except dynamic dispatch. Parameters are always passed by value in JavaScript/TypeScript, so the parameter passing modes in JAVASCRIPTY is an extension beyond JavaScript/TypeScript to illustrate a language design decision. We will update our type checker and small-step interpreter from Lab 4 and see that mutation forces to do a rather global refactoring of our interpreter.

We will also be exposed to the idea of transforming code to a "lowered" form to make it easier to implement interpretation.

Instructions. For this lab, find a new lab partner: you cannot work with your partners from Labs 1 to 4. You must work on this assignment in pairs. However,note that **each student is individually responsible for completing the assignment**.

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 will be working on the following three files:

- `Lab5.scala` with your answers to the coding exercises
- `Lab5Spec.scala` with any updates to your unit tests.
- `Lab5.jsy` with a challenging test case for your JAVASCRIPTY interpreter.

Be prepared to demonstrate and fully discuss your code in your interview. You will also be able to demonstrate that you completed the assignment in time by verifying your secure hash.

expressions	e	$::= x \mid n \mid b \mid \textbf{undefined} \mid uop\, e_1 \mid e_1\ bop\ e_2 \mid e_1\ ?\ e_2 : e_3$
		$\mid mut\ x = e_1; e_2 \mid \textbf{console.log}(e_1)$
		$\mid str \mid \textbf{function}\ p(params)\, tann\ e_1 \mid e_0(\overline{e})$
		$\mid \{\overline{f : e}\} \mid e_1.f \mid e_1 = e_2 \mid a \mid \textbf{null}$
		$\mid \textbf{interface}\ T\ \{\overline{f : \tau}\}\ ;\ e_1$
values	v	$::= n \mid b \mid \textbf{undefined} \mid str \mid \textbf{function}\ p(params)\, tann\ e_1$
		$\mid a \mid \textbf{null}$
location expressions	le	$::= x \mid e_1.f$
location values	lv	$::= *\, a \mid a.f$
unary operators	uop	$::= - \mid ! \mid * \mid \langle\tau\rangle$
binary operators	bop	$::= , \mid + \mid - \mid * \mid / \mid === \mid !== \mid < \mid <= \mid > \mid >= \mid \&\& \mid \mid\mid$
types	τ	$::= \textbf{number} \mid \textbf{bool} \mid \textbf{string} \mid \textbf{Undefined} \mid (params) \Rightarrow \tau' \mid \{\overline{f : \tau}\}$
		$\mid \textbf{Null} \mid T \mid \textbf{Interface}\ T\ \tau$
variables	x	
numbers (doubles)	n	
booleans	b	$::= \textbf{true} \mid \textbf{false}$
strings	str	
function names	p	$::= x \mid \varepsilon$
function parameters	$params$	$::= \overline{x : \tau} \mid mode\ x : \tau$
field names	f	
type annotations	$tann$	$::= : \tau \mid \varepsilon$
mutability	mut	$::= \textbf{const} \mid \textbf{var}$
passing mode	$mode$	$::= \textbf{name} \mid \textbf{var} \mid \textbf{ref}$
addresses	a	
type variables	T	
type environments	Γ	$::= \cdot \mid \Gamma[mut\ x \mapsto \tau]$
memories	M	$::= \cdot \mid M[a \mapsto k]$
contents	k	$::= v \mid \{\overline{f : v}\}$

Figure 1: Abstract Syntax of JAVASCRIPTY

Getting Started Download the code pack `lab5.tar.gz` from Piazza in the lab 5 tab.

A suggested way to get familiar with Scala is to do some small lessons with Scala Koans (`http://www.scalakoans.org/`). In particular, take a look at AboutByNameParameter, which introduces both `Either` and pass-by-name parameters (also known as call-by-name), as well as AboutAdvancedOptions, AboutForExpressions, and AboutTraversables.

1. **JavaScripty Implementation**

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

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

$$\textbf{var}\ x = e_1; e_2$$

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. Expressions may be mutable variables or fields of objects. We make all fields of objects mutable as is the default in JavaScript.

Parameter Passing Modes. In this lab, we can annotate function parameters with **var**, **name**, or **ref** to specify a parameter passing mode. The annotation **var** says the parameter should be pass-by-value with an allocation for a new mutable parameter variable initialized the argument value. The **name** and **ref** annotations specify pass-by-name and pass-by-reference, respectively. In Lab 4, all parameters were pass-by-value with an immutable variable, conceptually a "**const**" parameter. This "pass-by" terms are defined by their respective DoCall rules in Figure 9.

For simplicity, we consider two kinds of function parameters *params* that are either a sequence of pass-by-value with immutable variables $\overline{x : \tau}$ (as before and conceptually "**const**" parameters) or a single parameter with one of the new parameter passing modes $mode\ x : \tau$. This choice is purely for pedagogical reasons so that you do not need to deal with parameter lists when thinking about parameter passing modes. From the programmer's perspective, this would be a bit strange, as one would expect to be able specify a parameter list with independent passing modes for each parameter. Note that the two kinds of function parameters are implemented via the Scala `Either` type in the AST nodes.

Aliasing. In JavaScript, objects are dynamically allocated on the heap and then referenced with an extra level of indirection through a heap address. This indirection 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.

To model allocation, object literals of the form $\{\overline{f : v}\}$ are no longer values, rather they evaluate to an address a, which are then values representing objects. Addresses a are included in program expressions e because they arise during evaluation. However, there is no way to explicitly write an address in the source program. Addresses are an example of an enrichment of program expressions as an intermediate form solely for evaluation.

Casting and Type Declarations In the previous lab, we carefully crafted a very nice situation where as long as the input program passed the type checker, then evaluation would be free of run-time errors. Unfortunately, there are often programs that we want to execute that we cannot completely check statically and must rely on some amount of dynamic (run-time) checking.

We want to re-introduce dynamic checking in a controlled manner, so we ask that the programmer include explicit casts, written $\langle\tau\rangle\, e$. Executing a cast may result in a dynamic type error but intentionally nowhere else. Our `step` implementation should only result in throwing `DynamicTypeError` when executing a cast. For simplicitly, we limit the expressivity of casts to between object types.

Object types become quite verbose to write everywhere, so we introduce type declarations for them:

$$\textbf{interface}\ T\ \tau\ ;\ e$$

that says declare at type name T defined to be type τ that is in scope in expression e. We limit τ to be an object type. We do not consider T and τ to be same type (i.e., conceptually using name type equality for type declarations), but we permit casts between them. This choice enables typing of recursive data structures, like lists and trees (called recursive types).

Lowering: Removing Interface Declarations Type names become burdensome to work with as-is (e.g., requiring an environment to remember the mapping between T and τ). Instead, we will simplify the implementation of our later phases by first getting rid of **interface** type declarations, essentially replacing τ for T in e. We do not quite do this replacement because **interface** type declarations may be recursive and instead replace T with a new type form **Interface** $T\ \tau$ that bundles the type name T with its definition τ. In **Interface** $T\ \tau$, the type variable T should be considered bound in this construct.

This "lowering" is implemented in the function

```
def removeInterfaceDecl(e: Expr): Expr
```

that is provided for you. This function is very similar to substitution, but instead of substituting for program variables x (i.e., `Var(`x`)`), we substitute for type variables T (i.e., `TVar(`T`)`). Thus, we need an environment that maps type variable names T to types τ (i.e., the `env` parameter of type `Map[String,Typ]`).

In the `removeInterfaceDecl` function, we need to apply this type replacement anywhere the JAVASCRIPTY programmer can specify a type τ. We implement this process by recursively walking over the structure of the input expression looking for places to apply the type replacement.

Finally, we remove interface type declarations

$$\textbf{interface}\ T\ \tau\ ;\ e$$

by extending the environment with $[\,T \mapsto \textbf{Interface}\ T\ \tau]$ and applying the replacement in e.

With objects allocated on the heap, we also introduce the **null** value, which enables pointer-based data structures. The **null** value is not directly assignable to something of object type. The **null** value has type **Null**, which we make castable to any object type. But there is a cost to this flexibility, with **null**, we have to introduce another run-time check. We add another kind of run-time error for null dereference errors, which we write as nullerror and implement in `step` by throwing `NullDereferenceError`.

In Figure 2, we show the updated and new AST nodes. Note that `Deref` and `Cast` are `Uops` (i.e., they are unary operators).

Type Checking The inference rules defining the typing judgment form are given in Figures 3, 4, and 5.

- Similar to before, we implement type inference with the function

  def `typeInfer(env: Map[String,(Mutability,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 `MVar`) and a type.

- The type inference should use a helper function

  def `castOk(t1: Typ, t2: Typ): Boolean`

  that you also need to complete. This function specifies when type `t1` can be casted to type `t2` and implements the judgment form $\tau_1 \rightsquigarrow \tau_2$ given in Figure 5.

A template for the `Function` case for `typeInfer` is provided that you may use if you wish.

Reduction. We also update `substitute` and `step` from Lab 4. A small-step operational semantics is given in Figures 7–10.

The small-step judgment form is now as follows:[1]

$$\langle M, e\rangle \longrightarrow \langle M', e'\rangle$$

that says informally, "In memory M, expression e steps to a new configuration with memory M' and expression e'." The memory M is a map from addresses a to contents k, which include values and object values. The presence of a memory M that gets updated during evaluation is the hallmark of *imperative computation*.

- The `step` function now has the following signature

  def `step(e: Expr): DoWith[Mem,Expr]`

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

```
/* Declarations */
case class Decl(mut: Mutability, x: String, e1: Expr, e2: Expr) extends Expr
  Decl(mut, x, e1, e2)   mut x = e1; e2
case class InterfaceDecl(tvar: String, tobj: Typ, e: Expr) extends Expr
  InterfaceDecl(T, τ, e)   interface T τ ; e

sealed abstract class Mutability
case object Const extends Mutability
  MConst   const
case object Var extends Mutability
  MVar   var

/* Addresses and Mutation */
case class Assign(e1: Expr, e2: Expr) extends Expr
  Assign(e1, e2)   e1 = e2
case object Null extends Expr
  Null   null
case class A(addr: Int) extends Expr
  A(...)   a
case object Deref extends Uop
  Deref   *

/* Functions */
type Params = Either[ List[(String,Typ)], (PMode,String,Typ) ]
case class Function(p: Option[String], paramse: Params, tann: Option[Typ],
                    e1: Expr) extends Expr
  Function(p, params, tann, e1)   function p(params) tann e1

sealed abstract class PMode
case object PName extends PMode
  PName   name
case object PVar extends PMode
  PVar   var
case object PRef extends PMode
  PRef   ref

/* Casting */
case class Cast(t: Typ) extends Uop
  Cast(τ)   ⟨τ⟩

/* Types */
case class TVar(tvar: String) extends Typ
  TVar(T)   T
case class TInterface(tvar: String, t: Typ) extends Typ
  TInterface(T, τ)   Interface T τ
```

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.

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

$$\textsc{TypeVar}\ \frac{}{\Gamma \vdash x : \Gamma(x)} \qquad \textsc{TypeNeg}\ \frac{\Gamma \vdash e_1 : \textbf{number}}{\Gamma \vdash -e_1 : \textbf{number}} \qquad \textsc{TypeNot}\ \frac{\Gamma \vdash e_1 : \textbf{bool}}{\Gamma \vdash !e_1 : \textbf{bool}} \qquad \textsc{TypeSeq}\ \frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1 , e_2 : \tau_2}$$

$$\textsc{TypeArith}\ \frac{\Gamma \vdash e_1 : \textbf{number} \qquad \Gamma \vdash e_2 : \textbf{number} \qquad bop \in \{+, -, *, /\}}{\Gamma \vdash e_1\ bop\ e_2 : \textbf{number}} \qquad \textsc{TypePlusString}\ \frac{\Gamma \vdash e_1 : \textbf{string} \qquad \Gamma \vdash e_2 : \textbf{string}}{\Gamma \vdash e_1 + e_2 : \textbf{string}}$$

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

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

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

$$\textsc{TypeAndOr}\ \frac{\Gamma \vdash e_1 : \textbf{bool} \qquad \Gamma \vdash e_2 : \textbf{bool} \qquad bop \in \{\&\&, ||\}}{\Gamma \vdash e_1\ bop\ e_2 : \textbf{bool}} \qquad \textsc{TypePrint}\ \frac{\Gamma \vdash e_1 : \tau_1}{\Gamma \vdash \textbf{console.log}(e_1) : \textbf{Undefined}}$$

$$\textsc{TypeIf}\ \frac{\Gamma \vdash e_1 : \textbf{bool} \qquad \Gamma \vdash e_2 : \tau \qquad \Gamma \vdash e_3 : \tau}{\Gamma \vdash e_1\ ?\ e_2 : e_3 : \tau} \qquad \textsc{TypeNumber}\ \frac{}{\Gamma \vdash n : \textbf{number}} \qquad \textsc{TypeBool}\ \frac{}{\Gamma \vdash b : \textbf{bool}} \qquad \textsc{TypeString}\ \frac{}{\Gamma \vdash str : \textbf{string}}$$

$$\textsc{TypeUndefined}\ \frac{}{\Gamma \vdash \textbf{undefined} : \textbf{Undefined}} \qquad \textsc{TypeObject}\ \frac{\Gamma \vdash e_i : \tau_i \qquad \text{(for all } i\text{)}}{\Gamma \vdash \{\ldots, f_i : e_i, \ldots\} : \{\ldots, f_i : \tau_i, \ldots\}} \qquad \textsc{TypeGetField}\ \frac{\Gamma \vdash e : \{\ldots, f : \tau, \ldots\}}{\Gamma \vdash e.f : \tau}$$

Figure 3: Typing of non-imperative primitives and objects of JAVASCRIPTY (no change from the previous lab).

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

$$\textsc{TypeDecl}\ \frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma[mut\, x \mapsto \tau_1] \vdash e_2 : \tau_2}{\Gamma \vdash mut\, x = e_1; e_2 : \tau_2}$$

$$\textsc{TypeFunction}\ \frac{\Gamma[\overline{\textbf{const}\, x \mapsto \tau}] \vdash e_1 : \tau'}{\Gamma \vdash\ \textbf{function}\ (\overline{x : \tau})\ e_1\ :\ (\overline{x : \tau}) \Rightarrow \tau'}$$

$$\textsc{TypeFunctionAnn}\ \frac{\Gamma[\overline{\textbf{const}\, x \mapsto \tau}] \vdash e_1 : \tau'}{\Gamma \vdash\ \textbf{function}\ (\overline{x : \tau}) : \tau'\ e_1\ :\ (\overline{x : \tau}) \Rightarrow \tau'}$$

$$\textsc{TypeRecFunction}\ \frac{\Gamma[\textbf{const}\, x_0 \mapsto \tau''][\overline{\textbf{const}\, x \mapsto \tau}] \vdash e_1 : \tau' \qquad \tau'' = (\overline{x : \tau}) \Rightarrow \tau'}{\Gamma \vdash\ \textbf{function}\ x_0 (\overline{x : \tau}) : \tau'\ e_1\ :\ \tau''}$$

$$\textsc{TypeFunctionMode}\ \frac{\Gamma[\text{mut}(mode)\, x \mapsto \tau] \vdash e_1 : \tau'}{\Gamma \vdash\ \textbf{function}\ (mode\, x : \tau)\ e_1\ :\ (mode\, x : \tau) \Rightarrow \tau'}$$

$$\textsc{TypeFunctionAnnMode}\ \frac{\Gamma[\text{mut}(mode)\, x \mapsto \tau] \vdash e_1 : \tau'}{\Gamma \vdash\ \textbf{function}\ (mode\, x : \tau) : \tau'\ e_1\ :\ (mode\, x : \tau) \Rightarrow \tau'}$$

$$\textsc{TypeRecFunctionMode}\ \frac{\Gamma[\textbf{const}\, x_0 \mapsto \tau''][\text{mut}(mode)\, x \mapsto \tau] \vdash e_1 : \tau'}{\Gamma \vdash\ \textbf{function}\ x_0 (mode\, x : \tau) : \tau'\ e_1\ :\ (mode\, x : \tau) \Rightarrow \tau'}$$

$$\begin{aligned} \text{mut}(\textbf{name}) &\stackrel{\text{def}}{=} \textbf{const} \\ \text{mut}(\textbf{var}) &\stackrel{\text{def}}{=} \textbf{var} \\ \text{mut}(\textbf{ref}) &\stackrel{\text{def}}{=} \textbf{var} \end{aligned}$$

Figure 4: Typing of objects and binding constructs of JAVASCRIPTY. There is no rule for expression form **interface** $T\ \tau$; e because it is translated away prior to type checking.

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

$$\textsc{TypeNull}\ \frac{}{\Gamma \vdash \mathbf{null} : \mathbf{Null}} \qquad \textsc{TypeAssignVar}\ \frac{\mathbf{var}\, x \mapsto \tau \in \Gamma \qquad \Gamma \vdash e : \tau}{\Gamma \vdash x = e : \tau} \qquad \textsc{TypeAssignField}\ \frac{\Gamma \vdash e_1 : \{\ldots, f : \tau, \ldots\} \qquad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1.f = e_2 : \tau}$$

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

$$\textsc{TypeCallNameVar}\ \frac{\Gamma \vdash e_1 : (\mathit{mode}\, x : \tau) \Rightarrow \tau' \qquad \Gamma \vdash e_2 : \tau \qquad \mathit{mode} \neq \mathbf{ref}}{\Gamma \vdash e_1(e_2) : \tau'} \qquad \textsc{TypeCallRef}\ \frac{\Gamma \vdash e_1 : (\mathbf{ref}\, x : \tau) \Rightarrow \tau' \qquad \Gamma \vdash \mathit{le}_2 : \tau}{\Gamma \vdash e_1(\mathit{le}_2) : \tau'}$$

Requires Additional Dynamic Checking

$$\textsc{TypeCast}\ \frac{\Gamma \vdash e_1 : \tau_1 \qquad \tau_1 \rightsquigarrow \tau}{\Gamma \vdash \langle \tau \rangle\, e_1 : \tau}$$

Elide Rules for Intermediate Expressions

rule for $* e_1$ rule for a

$\boxed{\tau_1 \rightsquigarrow \tau_2}$

$$\textsc{CastOkEq}\ \frac{}{\tau \rightsquigarrow \tau} \qquad \textsc{CastOkNull}\ \frac{}{\mathbf{Null} \rightsquigarrow \{\ldots\}} \qquad \textsc{CastOkObject}_{\uparrow}\ \frac{\tau_i = \tau'_i \qquad \text{(for all } 1 \leq i \leq n \leq m\text{)}}{\{\ldots, f_i : \tau_i, \ldots, f_n : \tau_n, \ldots, f_m : \tau_m\} \rightsquigarrow \{\ldots, f_i : \tau'_i, \ldots, f_n : \tau'_n\}}$$

$$\textsc{CastOkObject}_{\downarrow}\ \frac{\tau_i = \tau'_i \qquad \text{(for all } 1 \leq i \leq n \leq m\text{)}}{\{\ldots, f_i : \tau_i, \ldots, f_n : \tau_n\} \rightsquigarrow \{\ldots, f_i : \tau'_i, \ldots, f_n : \tau'_n, \ldots, f_m : \tau'_m\}} \qquad \textsc{CastOkRoll}\ \frac{\tau_1 \rightsquigarrow \tau'_2[\mathbf{Interface}\ T\ \tau'_2 / T]}{\tau_1 \rightsquigarrow \mathbf{Interface}\ T\ \tau'_2}$$

$$\textsc{CastOkUnroll}\ \frac{\tau'_1[\mathbf{Interface}\ T\ \tau'_1 / T] \rightsquigarrow \tau_2}{\mathbf{Interface}\ T\ \tau'_1 \rightsquigarrow \tau_2}$$

Figure 5: Typing of imperative and type casting constructs of JAVASCRIPTY.

```
sealed class DoWith[W,R](doer: W => (W,R)) {
  def apply(w: W) = doer(w)

  def map[B](f: R => B): DoWith[W,B] = new DoWith[W,B]({
    (w: W) => {
      val (wp, r) = doer(w)
      (wp, f(r))
    }
  })

  def flatMap[B](f: R => DoWith[W,B]): DoWith[W,B] = new DoWith[W,B]({
    (w: W) => {
      val (wp, r) = doer(w)
      f(r)(wp) // same as f(a).apply(s)
    }
  })
}

def doget[W]: DoWith[W,W] = new DoWith[W,W]({ w => (w, w) })
def doreturn[W,R](r: R): DoWith[W,R] = doget map { _ => r }
def domodify[W](f: W => W): DoWith[W,Unit] = doget flatMap {
  w => new DoWith[W,Unit]({ _ => (f(w), ()) })
}
```

Figure 6: The `DoWith` type.

The `DoWith[W,R]` type is defined for you and shown in Figure 6. The essence of `DoWith[W,R]` is that it encapsulates a function of type `W=>(W,R)`, which can be seen as a computation that returns a value of type `R` with an input-output state of type `W`. The `doer` field holds precisely a function of the type `W=>(W,R)`. Seeing `DoWith[Mem,Expr]` as an encapsulated `Mem=>(Mem,Expr)`, we see how the judgment form $\langle M, e\rangle \longrightarrow \langle M', e'\rangle$ corresponds to the signature of `step`.

Note that the change in the judgment form necessitates updating *all* rules—even those that do not involve imperative features as in Figure 7. Note that for these rules, the memory M is simply threaded through. The main advantage of using the encapsulated computation `DoWith[Mem,Expr]` is that this common-case threading is essentially put into the `DoWith` data structure. One can view `DoWith[Mem,Expr]` as a "collection" that holds an `Expr` (with a computation over `Mem`), and thus, we can define a `map` method that creates an updated `DoWith` "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).

Some rules require allocating fresh addresses. For example, DoObject specifies allocating a new address a and extending the memory mapping a to the object. The address a is stated to be fresh by the constraint that $a \notin \mathrm{dom}(M)$. In the implementation, you call `Mem.alloc(k)` to get a fresh address with the memory cell initialized to contents `k`.

One 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 lab, 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#, Python).

One 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. Our interpreter instead simply allocates memory for local variables when they are encountered (e.g., DoVar). It never deallocates, even though we know that with local variables, those memory cells become inaccessible by the program once the function returns. The key observation is that the traditional stack is not essential for local variables but rather is an optimization for automatic deallocation based on function call-and-return.

Call-By-Name. The final wrinkle in our interpreter is that call-by-name requires substituting an arbitrary expression into another expression. Thus, we must be careful to avoid free variable capture (cf., Notes 3.2). We did not have to consider this case before because we were only ever substituting values that did not have free variables.

[1] Technically, the judgment form is not quite as shown because of the presence of the run-time error "markers" typeerror and nullerror.

In this lab, you will need to modify your `substitute` function to avoid free variable capture. A function to rename bound variables is given that

```
def avoidCapture(avoidVars: Set[String], e: Expr): Expr
```

renames bound variables in `e` to avoid variables given in `avoidVars`. Note that you will also need to call the function

```
def freeVars(e: Expr): Set[String]
```

that computes the set of free variables of an expression.

Type Safety. There is delicate interplay between the casts that we permit statically with

$$\tau_1 \rightsquigarrow \tau_2$$

and the dynamic checks that we need to perform at run-time (i.e., in

$$\langle M, e\rangle \longrightarrow \langle M', e'\rangle$$

as with TypeErrorCastObj or NullErrorDeref).

We say that a static type system (e.g., our $\Gamma \vdash e : \tau$ judgement form) is *sound* with respect to an operational semantics (e.g., our $\langle M, e\rangle \longrightarrow \langle M', e'\rangle$) if whenever our type checker defined by our typing judgment says a program is well-typed, then our interpreter defined by our small-step semantics never gets stuck (i.e., never throws `StuckError`).

Note that if the equality checks $\tau_i = \tau'_i$ in the premises of CastOkObject$_\uparrow$ and CastOkObject$_\downarrow$ were changed slightly to cast ok checks (i.e., $\tau_i \rightsquigarrow \tau'_i$), then our type system would become unsound with respect to our current operational semantics. For **extra credit**, carefully explain why by giving an example expression that demonstrates the unsoundness. Then, carefully explain what run-time checking you would add to regain soundness. First, give the explanation in prose, and then, try to formalize it in our semantics (if the challenge excites you!).

Memory doesn't change
Unary(Neg, N(42)) -> N(-42)
Wen't from M -> M

What should we test in code?
"Do Neg" should "negate values"
in {

}

$\boxed{\langle M, e\rangle \longrightarrow \langle M', e'\rangle}$

$$\textsc{DoNeg}\ \frac{n' = -n}{\langle M, -n\rangle \longrightarrow \langle M, n'\rangle} \qquad \textsc{DoNot}\ \frac{b' = \neg b}{\langle M, !\,b\rangle \longrightarrow \langle M, b'\rangle} \qquad \textsc{DoSeq}\ \frac{}{\langle M, v_1\ , e_2\rangle \longrightarrow \langle M, e_2\rangle}$$

$$\textsc{DoArith}\ \frac{n' = n_1\ bop\ n_2 \qquad bop \in \{+, -, *, /\}}{\langle M, n_1\ bop\ n_2\rangle \longrightarrow \langle M, n'\rangle} \qquad \textsc{DoPlusString}\ \frac{str' = str_1 + str_2}{\langle M, str_1 + str_2\rangle \longrightarrow \langle M, str'\rangle}$$

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

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

$$\textsc{DoOrTrue}\ \frac{}{\langle M, \textbf{true}\ ||\ e_2\rangle \longrightarrow \langle M, \textbf{true}\rangle} \qquad \textsc{DoOrFalse}\ \frac{}{\langle M, \textbf{false}\ ||\ e_2\rangle \longrightarrow \langle M, e_2\rangle} \qquad \textsc{DoPrint}\ \frac{v_1 \text{ printed}}{\langle M, \textbf{console.log}(v_1)\rangle \longrightarrow \langle M, \textbf{undefined}\rangle}$$

$$\textsc{DoIfTrue}\ \frac{}{\langle M, \textbf{true}\ ?\ e_2 : e_3\rangle \longrightarrow \langle M, e_2\rangle} \qquad \textsc{DoIfFalse}\ \frac{}{\langle M, \textbf{false}\ ?\ e_2 : e_3\rangle \longrightarrow \langle M, e_3\rangle} \qquad \textsc{SearchUnary}\ \frac{\langle M, e_1\rangle \longrightarrow \langle M', e_1'\rangle}{\langle M, uop\ e_1\rangle \longrightarrow \langle M', uop\ e_1'\rangle}$$

$$\textsc{SearchBinary}_1\ \frac{\langle M, e_1\rangle \longrightarrow \langle M', e_1'\rangle}{\langle M, e_1\ bop\ e_2\rangle \longrightarrow \langle M', e_1'\ bop\ e_2\rangle} \qquad \textsc{SearchBinary}_2\ \frac{\langle M, e_2\rangle \longrightarrow \langle M', e_2'\rangle}{\langle M, v_1\ bop\ e_2\rangle \longrightarrow \langle M', v_1\ bop\ e_2'\rangle}$$

$$\textsc{SearchPrint}\ \frac{\langle M, e_1\rangle \longrightarrow \langle M', e_1'\rangle}{\langle M, \textbf{console.log}(e_1)\rangle \longrightarrow \langle M', \textbf{console.log}(e_1')\rangle} \qquad \textsc{SearchIf}\ \frac{\langle M, e_1\rangle \longrightarrow \langle M', e_1'\rangle}{\langle M, e_1\ ?\ e_2 : e_3\rangle \longrightarrow \langle M', e_1'\ ?\ e_2 : e_3\rangle}$$

Figure 7: Small-step operational semantics of non-imperative primitives of JAVASCRIPTY. The only change compared to the previous lab is the threading of the memory.

$\boxed{\langle M, e\rangle \longrightarrow \langle M', e'\rangle}$

DoObject
$$\frac{a \notin \mathrm{dom}(M)}{\langle M, \{\overline{f : v}\}\rangle \longrightarrow \langle M[a \mapsto \{\overline{f : v}\}], a\rangle}$$

DoGetField
$$\frac{M(a) = \{\ldots, f : v, \ldots\}}{\langle M, a.f\rangle \longrightarrow \langle M, v\rangle}$$

SearchObject
$$\frac{\langle M, e_i\rangle \longrightarrow \langle M', e_i'\rangle}{\langle M, \{\ldots, f_i : e_i, \ldots\}\rangle \longrightarrow \langle M', \{\ldots, f_i : e_i', \ldots\}\rangle}$$

SearchGetField
$$\frac{\langle M, e_1\rangle \longrightarrow \langle M', e_1'\rangle}{\langle M, e_1.f\rangle \longrightarrow \langle M', e_1'.f\rangle}$$

DoConst
$$\frac{}{\langle M, \mathbf{const}\ x = v_1; e_2\rangle \longrightarrow \langle M, e_2[v_1/x]\rangle}$$

DoVar
$$\frac{a \notin \mathrm{dom}(M)}{\langle M, \mathbf{var}\ x = v_1; e_2\rangle \longrightarrow \langle M[a \mapsto v_1], e_2[*\,a/x]\rangle}$$

DoDeref
$$\frac{a \in \mathrm{dom}(M)}{\langle M, *\,a\rangle \longrightarrow \langle M, M(a)\rangle}$$

SearchDecl
$$\frac{\langle M, e_1\rangle \longrightarrow \langle M', e_1'\rangle}{\langle M, mut\ x = e_1; e_2\rangle \longrightarrow \langle M', mut\ x = e_1'; e_2\rangle}$$

DoAssignVar
$$\frac{a \in \mathrm{dom}(M)}{\langle M, *\,a = v\rangle \longrightarrow \langle M[a \mapsto v], v\rangle}$$

DoAssignField
$$\frac{M(a) = \{\ldots, f : v, \ldots\}}{\langle M, a.f = v'\rangle \longrightarrow \langle M[a \mapsto \{\ldots, f : v', \ldots\}], v'\rangle}$$

SearchAssign$_1$
$$\frac{\langle M, e_1\rangle \longrightarrow \langle M', e_1'\rangle \qquad e_1 \neq lv_1}{\langle M, e_1 = e_2\rangle \longrightarrow \langle M', e_1' = e_2\rangle}$$

SearchAssign$_2$
$$\frac{\langle M, e_2\rangle \longrightarrow \langle M', e_2'\rangle}{\langle M, lv_1 = e_2\rangle \longrightarrow \langle M', lv_1 = e_2'\rangle}$$

DoCall
$$\frac{v = \mathbf{function}\ (x_1 : \tau_1, \ldots, x_n : \tau_n)\, tann\ e}{\langle M, v(v_1, \ldots v_n)\rangle \longrightarrow \langle M, e[v_n/x_n]\cdots[v_1/x_1]\rangle}$$

DoCallRec
$$\frac{v = \mathbf{function}\ x(x_1 : \tau_1, \ldots, x_n : \tau_n)\, tann\ e}{\langle M, v(v_1, \ldots v_n)\rangle \longrightarrow \langle M, e[v_n/x_n]\cdots[v_1/x_1][v/x]\rangle}$$

SearchCall$_1$
$$\frac{\langle M, e\rangle \longrightarrow \langle M', e'\rangle}{\langle M, e(e_1, \ldots, e_n)\rangle \longrightarrow \langle M', e'(e_1, \ldots, e_n)\rangle}$$

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

Figure 8: Small-step operational semantics of objects, binding constructs, variable and field assignment, and function call of JavaScript.

$\boxed{\langle M, e\rangle \longrightarrow \langle M', e'\rangle}$

DoCallName
$$\frac{v = \textbf{function}\ (\textbf{name}\ x_1 : \tau)\, tann\, e_1}{\langle M, v(e_2)\rangle \longrightarrow \langle M, e_1[e_2/x_1]\rangle}$$

DoCallRecName
$$\frac{v = \textbf{function}\ x(\textbf{name}\ x_1 : \tau)\, tann\, e_1}{\langle M, v(e_2)\rangle \longrightarrow \langle M, e_1[e_2/x_1][v/x]\rangle}$$

DoCallVar
$$\frac{v = \textbf{function}\ (\textbf{var}\ x_1 : \tau)\, tann\, e_1 \qquad a \notin \mathrm{dom}(M)}{\langle M, v(v_2)\rangle \longrightarrow \langle M[a \mapsto v_2], e_1[* a/x_1]\rangle}$$

DoCallRecVar
$$\frac{v = \textbf{function}\ x(\textbf{var}\ x_1 : \tau)\, tann\, e_1 \qquad a \notin \mathrm{dom}(M)}{\langle M, v(v_2)\rangle \longrightarrow \langle M[a \mapsto v_2], e_1[* a/x_1][v/x]\rangle}$$

DoCallRef
$$\frac{v = \textbf{function}\ (\textbf{ref}\ x_1 : \tau)\, tann\, e_1}{\langle M, v(lv_2)\rangle \longrightarrow \langle M, e_1[lv_2/x_1]\rangle}$$

DoCallRecRef
$$\frac{v = \textbf{function}\ x(\textbf{ref}\ x_1 : \tau)\, tann\, e_1}{\langle M, v(lv_2)\rangle \longrightarrow \langle M, e_1[lv_2/x_1][v/x]\rangle}$$

SearchCallVar
$$\frac{\langle M, e_2\rangle \longrightarrow \langle M', e_2'\rangle}{\langle M, \big(\textbf{function}\ p(\textbf{var}\ x : \tau)\ e_1\big)(e_2)\rangle \longrightarrow \langle M', \big(\textbf{function}\ p(\textbf{var}\ x : \tau)\ e_1\big)(e_2')\rangle}$$

SearchCallRef
$$\frac{\langle M, e_2\rangle \longrightarrow \langle M', e_2'\rangle \qquad e_2 \neq lv_2}{\langle M, \big(\textbf{function}\ p(\textbf{ref}\ x : \tau)\ e_1\big)(e_2)\rangle \longrightarrow \langle M', \big(\textbf{function}\ p(\textbf{ref}\ x : \tau)\ e_1\big)(e_2')\rangle}$$

Figure 9: Small-step operational semantics of function call with parameter passing modes of JavaScripty.

$\boxed{\langle M, e\rangle \longrightarrow \langle M', e'\rangle}$

DoCast
$$\frac{v \neq \textbf{null} \qquad v \neq a}{\langle M, \langle \tau\rangle\, v\rangle \longrightarrow \langle M, v\rangle}$$

DoCastNull
$$\frac{\tau = \{\ldots\} \text{ or } \textbf{Interface}\ T\ \{\ldots\}}{\langle M, \langle \tau\rangle\, \textbf{null}\rangle \longrightarrow \langle M, \textbf{null}\rangle}$$

DoCastObj
$$\frac{M(a) = \{\ldots\} \qquad \tau = \{\ldots, f_i : \tau_i, \ldots\} \text{ or } \textbf{Interface}\ T\ \{\ldots, f_i : \tau_i, \ldots\} \qquad f_i \in \mathrm{dom}(M(a)) \qquad \text{for all } i}{\langle M, \langle \tau\rangle\, a\rangle \longrightarrow \langle M, a\rangle}$$

TypeErrorCastObj
$$\frac{M(a) = \{\ldots\} \qquad \tau = \{\ldots, f_i : \tau_i, \ldots\} \text{ or } \textbf{Interface}\ T\ \{\ldots, f_i : \tau_i, \ldots\} \qquad f_i \notin \mathrm{dom}(M(a)) \qquad \text{for some } i}{\langle M, \langle \tau\rangle\, a\rangle \longrightarrow \mathsf{typeerror}}$$

NullErrorGetField
$$\frac{}{\langle M, \textbf{null}.f\rangle \longrightarrow \mathsf{nullerror}}$$

NullErrorAssignField
$$\frac{}{\langle M, \textbf{null}.f = e\rangle \longrightarrow \mathsf{nullerror}}$$

typeerror and nullerror propagation rules elided

Figure 10: Small-step operational semantics of type casting and null dereference errors of JavaScripty.