CSCI 3155: Lab Assignment 5

Checkpoint: Friday, March 18, 2016 at 6pm
Due Date: Friday, April 1, 2016 at 6pm

The primary purpose of this lab to explore mutation or imperative updates in programming languages. With mutation, we explore two related language considerations: parameter passing modes and casting. Concretely, we extend JAVASCRIPTY with mutable variables and objects, parameter passing modes, (recursive) type declarations, type casting. 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. In particular, we consider parameter passing modes primarily to illustrate a language design decision and how the design decision manifests in the operational semantics. Call-by-value with addresses and call-by-reference are often confused, but with the operational semantics, we can see clearly the distinction.

We will update our type checker and small-step interpreter from Lab 4 and see that mutation forces a global refactoring of our interpreter. To minimize the impact of this refactoring, we will be explore the functional programming idea of encapsulating computation in a data structure (known as a *monad*). We will also consider the idea of transforming code to a "lowered" form to make it easier to implement interpretation.

PL Ideas Imperative programming (memory, addresses, aliasing). Language design choices (via parameter passing modes as a case study).

FP Skills Encapsulating computation as a data structure.

Instructions. Find a new partner for this lab assignment (different from your previous partners). You will work on this assignment closely with your partner. However, note that **each student needs to submit** and are 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.

Similar to previous lab assignments, we have a code **checkpoint**. This is to encourage you to start the coding portion of the assignment early and it requires you to submit a partial solution on COG early. The minimum grade for the checkpoint will be announced in lectures and recitation. **Failure to submit the checkpoint will result in a maximum of 80% on your final grade for this lab assignment.** Additionally, there is now a cap on the number of submissions (see **COG submission policy** below).

Recall the evaluation guideline from the course syllabus.

> *Both your ideas and also the clarity with which they are expressed matter—both in your English prose and your code!*
>
> *We will consider the following criteria in our grading:*
>
> - How well does your submission answer the questions? *For example, a common mistake is to give an example when a question asks for an explanation. An example may be useful in your explanation, but it should not take the place of the explanation.*
> - How clear is your submission? *If we cannot understand what you are trying to say, then we cannot give you points for it. Try reading your answer aloud to yourself or a friend; this technique is often a great way to identify holes in your reasoning. For code, not every program that "works" deserves full credit. We must be able to read and understand your intent. Make sure you state any pre-conditions or invariants for your functions (either in comments, as assertions, or as* `require` *clauses as appropriate).*

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 unfamiliar to you.

Finally, make sure that your file compiles and runs on COG. A program that does not compile will *not* be graded. We have also instituted a **new COG submission policy**: for each of the 1-week cycles — checkpoint, final — up to 10 submissions can be made to COG. Subsequent submissions will result in a 2% penalty per submission. Example: submitting 35 times will result in a 50% penalty on the autograded score.

Submission Instructions. Upload to Moodle exactly four files named as follows:

- `Lab5_`*YourIdentiKey*`.scala` with your answers to the coding exercises
- `Lab5Spec-`*YourIdentiKey*`.scala` with any updates to your unit tests.
- `Lab5-`*YourIdentiKey*`.jsy` with a challenging test case for your JAVASCRIPTY interpreter.
- `Lab5_`*YourIdentiKey*`.pdf` with your answers to the written exercises (only written exercise is for extra credit).

Replace *YourIdentiKey* with your IdentiKey (e.g., I would submit `Lab5-bec.pdf` and so forth). Don't use your student identification number. To help with managing the submissions, we ask that you rename your uploaded files in this manner.

Submit your `Lab5.scala` file to COG for auto-testing. We ask that you submit both to COG and to Moodle in case of any issues.

Sign-up for an interview slot for an evaluator. To fairly accommodate everyone, the interview times are strict and **will not be rescheduled**. Missing an interview slot means missing the interview evaluation component of your lab grade. Please take advantage of your interview time to maximize the feedback that you are able receive. Arrive at your interview ready to show

```
sealed class DoWith[W,R] private (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(r).apply(wp)
    }
  })
}

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

Figure 1: The `DoWith` type.

your implementation and your written responses. Implementations that do not compile and run will not be evaluated.

Getting Started. Clone the code from the Github repository with the following command:

```
git clone -b lab5 https://github.com/bechang/pppl-labs.git  lab5
```

A suggested way to get familiar with Scala is to do some small lessons with Scala Koans (`http://www.scalakoans.org/`).

1. **Feedback**. Complete the survey on the linked from Moodle after completing this assignment. Any non-empty answer will receive full credit.

2. **Warm-Up: Encapsulating Computation**. To implement our interpreter for JAVASCRIPTY with memory, we introduce the idea of encapsulating computation with the `DoWith[W,R]` type. This idea builds on the concepts of abstract data types, collections, and higher-order functions introduced in Lab 4.

   The `DoWith` type constructor is defined for you in the `jsy.lab5` package and shown in Figure 1. The essence of the `DoWith[W,R]` type is that it encapsulates a function of type `W=>(W,R)`, which is 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)`.

We should view `DoWith[W,R]` as a "collection" somewhat like `List[A]`. Recall that a value of type `List[A]` encapsulates a sequence of elements of type `A`; it also has methods to process and transform those elements. Similarly, a value of type `DoWith[W,R]` encapsulates a computation *with* an input-output state `W` for a result `R`; it also has methods to process and transform that computation.

Consider the `map` method shown in Figure 1. Let us focus on the signature of the `map` method:

```
class DoWith[W,R] { def map[B](f: R => B): DoWith[W,B] }
```

From the signature, we see that the `map` method transforms a `DoWith` holding a computation *with* a `W` for a `R` to one for a `B` using the callback `f`. Intuitively, the input computation (bound to **this**) has the result *r* : `R`. Using `map` transforms it to computation that will yield the result `f(`*r*`)`:`B`.

The `flatMap` method has a signature that is quite similar to `map`:

```
class DoWith[W,R] { def flatMap[B](f: R => DoWith[W,B]): DoWith[W,B] }
```

But note there's a difference: `flatMap` allows the callback `f` to return a `DoWith[W,B]` computation. Intuitively, `flatMap` sequences the input computation (bound to **this**) that will yield a result *r* : `R` with the computation obtained from `f(`*r*`)`.

We also have four functions `doget`, `doput`, `doreturn`, and `domodify` for constructing `DoWith` objects. (We disallow the direct construction of `DoWith` objects, by using the `private` modifier.)

- `doget` creates a computation whose result is the current state `w`.
- `doput[W](w:W)` creates a computation that sets the state to `w` (and whose result is just unit `()`).
- `doreturn[W,R](r:R)` creates a computation that leaves the state untouched, but whose result is `r`.
- `domodify[W](f:W => W)` creates a computation that modifies the state according to `f`.

Note that the `doreturn` and `domodify` functions are not strictly needed, because they can be defined in terms of `doget`, `doput`, `map`, and `flatMap`. But we provide them because they are commonly-needed operations.

In this warmup question, we practice using the `DoWith[W,R]` type.

- Implement a function

  ```
  def rename(env: Map[String, String], e: Expr): DoWith[Int,Expr]
  ```

  that yields a computation to yield a resulting expression that is a version of the input expression `e` with bound variables renamed. The environment `env` maps names for free variables in `e` to what they should be renamed to in the result. You should use the provided helper function

  ```
  def fresh: DoWith[Int,String]
  ```

for creating fresh variable names.

For the purposes of this exercise, we will only consider the subset of the expression language, consisting of the following:

$$e ::= x \mid n \mid e_1 + e_2 \mid \textbf{const}\ x = e_1; e_2$$

The strategy that we will use for renaming will be to globally renaming all variables uniquely using an integer counter. For example, we will rename

```
const a  = (const a  = 1; a ); (const a  = 2; a )
```

to

```
const x0 = (const x1 = 1; x1); (const x2 = 2; x2) .
```

Note that we will completely ignore the names given in the input, so the following expression will also be renamed to the syntactically same expression as above:

```
const a  = (const b  = 1; b ); (const c  = 2; c ) .
```

This policy for new variable names is captured in given helper function `fresh`, and we will rename expressions from left-to-right.

Seeing the `DoWith[Int,Expr]` type as an encapsulated `Int=>(Int,Expr)`, we see that the signature our `rename` is conceptually

```
def rename(env: Map[String, String], e: Expr): (Int => (Int, Expr))
```

The `rename` function is thus conceptually a curried function that takes as input first `env` and `e`, which returns a function that takes an integer i to return a integer-expression pair (i', e'). The integer state captures the next available variable number.

Hint: The only functions or methods for manipulating `DoWith` objects in this exercise are `doreturn`, `map`, and `flatMap`. The `doget` and `doput` functions are used in `fresh`, but you will not need to call them directly.

3. **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 2 and the new AST nodes are given in Figure 3.

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

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

and then include an assignment expression:

$$e_1 = e_2$$

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	$::= - \mid \texttt{!} \mid * \mid \langle\tau\rangle$
binary operators	bop	$::= \texttt{,} \mid \texttt{+} \mid \texttt{-} \mid \texttt{*} \mid \texttt{/} \mid \texttt{===} \mid \texttt{!==} \mid \texttt{<} \mid \texttt{<=} \mid \texttt{>} \mid \texttt{>=} \mid \texttt{\&\&} \mid \texttt{||}$
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 2: Abstract Syntax of JAVASCRIPTY

```
/* 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 3: 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.

that writes the value of e_2 to a location named by expression e_1. Expressions may be mutable variables x or fields of objects $e_1.f$. We make all fields of objects mutable as is the default in JavaScript.

We remove the AST node `ConstDecl` and replace it with the node `Decl` with an additional parameter `mut : Mutability` that specifies the mutability of the variable, that is, whether the variable is **const** or **var**.

Aliasing. In JavaScript and in this lab, 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., they are bound to the same object value). 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 the values representing objects (as shown below):

$$\text{values} \quad v ::= \ldots \mid a \mid \textbf{null}$$

With objects allocated on the heap, we also introduce the **null** value (or often called the **null** pointer). We consider an unbounded set of addresses that is disjoint from the **null** value (i.e., a cannot stand for **null**). Addresses a are also 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 to express small-step evaluation.

Parameter Passing Modes. We can annotate function parameters with **var**, **ref**, or **name** to specify a parameter passing mode. The annotation **var** says the parameter should be call-by-value with an allocation for a new mutable parameter variable initialized the argument value. The **ref** and annotations specify call-by-reference and call-by-name, respectively. In Lab 4, all parameters were call-by-value with an immutable variable, conceptually a "**const**" parameter. This "call-by" terms are defined by their respective DoCall rules in Figure 9. The intellectual exercise here is to decode what these "call-by" terms mean by reading their respective rules. Observe from the rules that the **ref** requires an intermediate language with addresses (and mutation to be interesting), but **name** could be a useful language feature in a pure setting. Call-by-name is a specific instance of *lazy evaluation*.

To simplify the lab implementation, we consider in the syntax 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. In the AST nodes, the two kinds of function parameters are implemented via the Scala `Either` type (see Figure 3).

Casting. 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 simplicity, we limit the expressivity of casts to between object types.

The **null** value has type **Null** and is not directly assignable to something of object type, but we make **Null** castable to any object type. However, 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`.

Abstract Syntax Trees. In Figure 3, we show the updated and new AST nodes. Note that `Deref` and `Cast` are `Uops` (i.e., they are unary operators).

(a) **Exercise: Type Checking.** The inference rules defining the typing judgment form are given in Figures 4, 5, and 6.

- 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 6.

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

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

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

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

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

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

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

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

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

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

Figure 4: Typing of non-imperative primitives and objects of JAVASCRIPTY (no change from the previous lab).

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

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

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

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

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

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

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

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

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

Figure 5: Typing of objects and binding constructs of JAVASCRIPTY.

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

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

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

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

Requires Additional Dynamic Checking

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

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

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

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

Figure 6: Typing of imperative and type casting constructs of JAVASCRIPTY.

(b) **Exercise: Reduction.** We also update `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.*

Note that the change in the judgment form necessitates updating *all* rules—even those that do not involve imperative features as in Figure 7. For these rules, the memory M is simply threaded through (see Figure 7).

- 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.

Seeing the `DoWith[Mem,Expr]` type 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`. In particular, the signature our `step` is conceptually

def `step(e: Expr): (Mem => (Mem, Expr))`

The `step` function is thus conceptually a curried function that takes as input first e, which returns a function that takes M to return (M', e').

The Crucial Observation. The main advantage of using the encapsulated computation type `DoWith[Mem,Expr]` is that we can put this common-case threading into the `DoWith` data structure.

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 `memalloc(k)` to get a fresh address with the memory cell initialized to contents `k`.

(c) **Exercise: 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.

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`

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

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

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

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

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

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

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

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

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

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

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.

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.

Memory. 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.

Type Safety. There is a 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!).

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

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 JavaScripty.

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

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

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

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. Ignore the "or **Interface** ..." parts unless attempting the extra credit implementation.

4. **Extra Credit: Type Declarations and Recursive Types**.

   This exercise is for extra credit. Please only attempt this exercise if you have fully completed the rest of the lab.

   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).

   (a) **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" should be implemented in the function

   def `removeInterfaceDecl(e: Expr): Expr`

   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.

   (b) **Updating Type Checking and Reduction**

   To update type checking with interface declarations, we only need to update `castOk` with the rules given in Figure 11.

   The update to `step` is also quite small. We only need to update a few cases corresponding to casting shown in Figure 10.

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

$$\textsc{CastOkRoll}\ \frac{\tau_1 \rightsquigarrow \tau_2'[\textbf{Interface}\ T\ \tau_2'/T]}{\tau_1 \rightsquigarrow \textbf{Interface}\ T\ \tau_2'} \qquad \textsc{CastOkUnroll}\ \frac{\tau_1'[\textbf{Interface}\ T\ \tau_1'/T] \rightsquigarrow \tau_2}{\textbf{Interface}\ T\ \tau_1' \rightsquigarrow \tau_2}$$

Figure 11: Type casting with interfaces.