CSCI 3155: Lab Assignment 5

Fall 2016
Checkpoint due Friday, October 28, 2016
Assignment due Friday, November 4, 2016

Learning Goals. The primary learning goals of this lab are to understand the following:

- imperative computation;
- mutation and aliasing;
- casting and type safety;
- recursive types; and
- programming with encapsulated effects.

PL Ideas Imperative computation (memory, addresses, aliasing). Casting and type safety. Recursiive types.

FP Skills Encapsulating computation as a data structure.

Concretely, 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 effects 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.

Extending our discussion about parameter passing modes to illustrate language design decisions and how design decisions manifest 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.

General Guidelines. During recitation find a partner for this lab assignment (should be different for every lab assignment). 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 beyond your teams. However, we ask that you code in pairs. See the collaboration policy for details, including the following:

> *Bottom line, feel free to use resources that are available to you as long as the use is **reasonable** and you **cite** them in your submission. However, copying answers directly or indirectly from solution manuals, web pages, or your peers is certainly unreasonable.*

Also, 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 via `sbt test`. A program that does not compile will *not* be graded—no interview will be conducted.

Submission Instructions. We are using Github for assignment distribution and submission. You need to have a Github identity and must have your full name in your Github profile so that we can associate you with your submissions.

You will be editing and submitting the the following files:

- `src/main/scala/jsy/student/Lab5.scala` with your solution to the coding exercises;
- `src/test/scala/jsy/student/Lab5Spec.scala` with your additional tests; and
- `lab5-`*yourteamname*`.jsy` with a challenging test case for your JAVASCRIPTY interpreter.

You are also likely to edit `src/main/scala/jsy/student/Lab5Worksheet.sc` for any scratch work.

Following good git practice, please make commits in small bits corresponding to completing small conceptual parts and push often so that your progress is evident. We expect that you have some familarity with git from prior courses. If not, please discuss with your classmates and the course staff (e.g., via Piazza).

At any point, you may submit your `Lab5.scala` file to COG for auto-testing. You need to submit to COG for the auto-testing part of your score, as well as to continue to the interview.

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 score. Please take advantage of your interview time

to maximize the feedback that you are able receive. Arrive at your interview ready to show your team's implementation and your written responses. Implementations that do not compile and run will not be evaluated.

Finally, upload to the moodle exactly the files named above, that is,

- `Lab5.scala`
- `Lab5Spec.scala`
- `lab5-`*`yourteamname`*`.jsy`

Getting Started. First, form a team of two and pick a team name. For our bookkeeping, please prefix your team name with `lab5-` (e.g., `lab5-anatomists`).

You must work in teams of two, and you will form teams in lab section. If you miss lab section on the day teams are formed, you need to find a partner on your own. If you really, really cannot find a partner, then please contact the course staff (via Piazza).

Then, log into moodle and follow the Github Classroom link for setting up your Lab 5 repository with your team name. The first person will create the team, and the second person will select the team name from the existing team names.

If you would like to look at the code before getting your own copy for submission, you may go to `https://github.com/csci3155/pppl-lab5`.

Checkpoint. The checkpoint is to encourage you to start the coding portion of the assignment early and it requires you to submit your partial solution on COG a week before the assignment is due. You do not need to complete all coding a week early but we want you to start working on it. This means that submitting the empty template that fails all tests is **not sufficient**. Failing to submit to the checkpoint will prevent you from proceeding to the interview. However, as long as you pass the checkpoint, this early score from the checkpoint will not affect your grade for the assignment or your overall grade for the course.

Scala Practice. 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 the 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

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

Figure 1: The `DoWith` type.

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.

(a) **Exercise: Update mapFirst to mapFirstWith.** Update the `mapFirst` function Lab 4 that finds the first element in `l` where `f` applied to it returns a `Some`(d) for some value d : `DoWith[W,A]`. It should return a `DoWith[W,List[A]]` that contains `l` with that element replaced with the contents of d.

```
def mapFirstWith[W,A](l: List[A])(f: A => Option[DoWith[W,A]])
      : DoWith[W,List[A]]
```

(b) **Exercise: Implement mapWith.** Implement a version of `map` for `List` and `Map` that takes a mapping function `f` that returns a `DoWith[W,B]` instead of `B`:

```
def mapWith[W,A,B](l: List[A])(f: A => DoWith[W,B])
      : DoWith[W,List[B]]
def mapWith[W,A,B,C,D](m: Map[A,B])(f: ((A,B)) => DoWith[W,(C,D)])
      : DoWith[W,Map[C,D]]
```

These functions apply `f` to each element of `l` or `m` from the right sequencing the resulting `DoWith`s to construct the mapped `List` or `Map` respectively. You will find these functions useful for implementing `rename`.

(c) **Exercise: Rename with DoWith.** Port your rename function from Lab 4:

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

that yields a computation to yield a resulting expression that is a version of the input expression `e` with bound variables renamed according to `fresh`. The environment `env` maps original names to new names for free variables of `e`. The inner helper function

```
def ren(env: Map[String,String], e: Expr): DoWith[W,Expr]
```

recurses over expression `e` with such a renaming environment `env`. **Hint**: The only functions or methods for manipulating `DoWith` objects needed in this exercise are `doreturn`, `map`, and `flatMap`.

Looking at how a client could use your rename function, one could, for example, globally renaming all variables uniquely using an integer counter for each name. For example, rename

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

to

```
const a0 = (const a1 = 1; a1); (const a2 = 2; a2) .
```

This policy is implemented by the given `uniquify` function by calling your `rename` function with a particular choice for the `fresh` parameter.

Implement an alternative policy that instead ignores the programmer-supplied variable names instead calls a variables `x`n for a unique number n, that is, the literal string `x` followed by a counter. So the above expression is renamed to

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

Since we will completely ignore the programmer-supplied given in the input, the following expression will also be renamed to the syntactically same expression as above:

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

To implement this policy,

(d) **Exercise: Fresh: Applying a DoWith.** In the lab template, implement the helper function `fresh:String => DoWith[Int,String]` for the `myuniquify` function. **Hint**: The only functions for manipulating `DoWith` objects needed in this exercise are `doget` and `doput`.

Looking at how `myuniquify` calls `rename` and then `ren`, we can build some intuition for how the `DoWith` data structure works. In this case, `W` is chosen to be `Int`, so the `ren` function returns `DoWith[Int,Expr]`.

A `DoWith[Int,Expr]` encapsulates a function of type `Int=>(Int,Expr)`. So conceptually, we can see the `ren` function as having the following signature:

```
def ren(env: Map[String,String], e: Expr):  Int => (Int,Expr)    or
def ren(env: Map[String,String], e: Expr)(i: 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.

3. **JavaScripty Implementation**

At this point, we are used to extending our interpreter implementation by updating our type checker `typeof` 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.

expressions	e	$::= x \mid n \mid b \mid \textbf{undefined} \mid uop\, e_1 \mid e_1\ bop\ e_2 \mid e_1\ ?\ e_2 : e_3$
		$\mid m\, x = e_1; e_2 \mid \textbf{console.log}(e_1)$
		$\mid str \mid p(\overline{x : \varsigma})\, tann \Rightarrow e_1 \mid e_0(\overline{e})$
		$\mid \{\overline{f : e}\} \mid e_1.f \mid e_1 = e_2 \mid a \mid \textbf{null}$
		$\mid \textbf{interface}\ T\ \{\overline{f : \tau}\}\ ;\ e_1$
values	v	$::= n \mid b \mid \textbf{undefined} \mid str \mid p(\overline{x : \varsigma})\, tann \Rightarrow e_1$
		$\mid a \mid \textbf{null}$
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	$::= \texttt{,} \mid + \mid - \mid * \mid / \mid \texttt{===} \mid \texttt{!==} \mid < \mid <= \mid > \mid >= \mid \&\& \mid \mid\mid$
types	τ	$::= \textbf{number} \mid \textbf{bool} \mid \textbf{string} \mid \textbf{Undefined} \mid (\overline{x : \varsigma}) \Rightarrow \tau \mid \{\overline{f : \tau}\}$
		$\mid \textbf{Null} \mid T \mid \textbf{Interface}\ T\ \tau$
moded types	ς	$::= m\tau$
parameter mode	m	$::= \textbf{const} \mid \textbf{name} \mid \textbf{var} \mid \textbf{ref}$
variables	x, y	
numbers (doubles)	n	
booleans	b	$::= \textbf{true} \mid \textbf{false}$
strings	str	
function names	p	$::= x \mid \varepsilon$
field names	f	
type annotations	$tann$	$::= : \tau \mid \varepsilon$
addresses	a	
type variables	T	
type environments	Γ	$::= \cdot \mid \Gamma[x \mapsto \varsigma]$
memories	M	$::= \cdot \mid M[a \mapsto k]$
contents	k	$::= v \mid \{\overline{f : v}\}$

Figure 2: Abstract Syntax of JAVASCRIPTY

```
/* Parameter Modes */
case object MVar extends Mode
  MVar  var
case object MRef extends Mode
  MRef  ref

/* 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  *

/* 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

/* Type Declarations */
case class InterfaceDecl(tvar: String, t: Typ, e: Expr) extends Expr
  InterfaceDecl(T, τ, e)  interface T τ ; e
```

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.

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 x or fields of objects $e_1.f$. We make all fields of objects mutable as is the default in JavaScript.

Parameter Passing Modes. We can now also annotate function parameters with **var** or **ref** to specify a parameter passing mode for mutable locations. 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** annotation specifies call-by-reference where the parameter location is shared or *aliased* with an existing memory location. These "call-by" terms are defined by their respective DoCall rules in Figure 8. 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 as in Lab 4.

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`.

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

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

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

  that you need to complete. Note that the type environment now maps a variable name to a pair of a mode and a type in order to check whether or not a variable can be assigned to.

- The type inference should use a helper function

  def `isBindex(m: Mode, e: Expr): Boolean`

  and

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

  that you also need to complete. The `isBindex` corresponds to the judgment form $m \vdash e$ bindex specifying when an expression is bindable to a variable under the mode m, and the `castOk` 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.

(b) **Exercise: Reduction.** We also update `step` from Lab 4. A small-step operational semantics is given in Figures 6–9.

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 6. For these rules, the memory M is simply threaded through (see Figure 6).

- 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, along with copying the `inequalityVal` and `iterate` functions and updating the `substitute` function.
- The following helper functions need to be implemented and are used by `step`:

  def `isRedex(m: Mode, e: Expr): Boolean`
  def `getBinding(m: Mode, e: Expr): DoWith[Mem,Expr]`

  that correspond to the $m \vdash e$ redex and $m \vdash \langle M, e\rangle \hookrightarrow \langle M', e'\rangle$, respectively. The first judgment form captures when expression e is reducible under a mode m (like in Lab 4). Then, in the case that e is *not* reducible under m, the second judgment form defines what expression e' should be used for binding (with a potentialy updated memory M').

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))`

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

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

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

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

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

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

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

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

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

$$\textsc{TypeFunction}\ \frac{\Gamma[\overline{x \mapsto \varsigma}] \vdash e_1 : \tau}{\Gamma \vdash\ (\overline{x : \varsigma}) => e_1\ :\ (\overline{x : \varsigma}) \Rightarrow \tau} \qquad \textsc{TypeFunctionAnn}\ \frac{\Gamma[\overline{x \mapsto \varsigma}] \vdash e_1 : \tau}{\Gamma \vdash\ (\overline{x : \varsigma}) : \tau => e_1\ :\ (\overline{x : \varsigma}) \Rightarrow \tau}$$

$$\textsc{TypeRecFunction}\ \frac{\Gamma[y \mapsto \tau'][\overline{x \mapsto \varsigma}] \vdash e_1 : \tau' \qquad \tau' = (\overline{x : \varsigma}) \Rightarrow \tau}{\Gamma \vdash\ y(\overline{x : \varsigma}) : \tau => e_1\ :\ \tau'}$$

Figure 4: Typing of non-imperative JAVASCRIPTY (minimal change from the previous lab).

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

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

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

$$\textsc{TypeCall}\ \frac{\Gamma \vdash e : (x_1 : m_1\,\tau_1, \ldots, x_n : m_n\,\tau_n) \Rightarrow \tau \qquad \Gamma \vdash e_i : \tau_i \qquad m_i \vdash e_i\ \mathsf{bindex} \qquad (\text{for all } 1 \le i \le n)}{\Gamma \vdash e(e_1, \ldots, e_n) : \tau}$$

$\boxed{m \vdash e\ \mathsf{bindex}}$

$$\textsc{ValBind}\ \frac{}{m \vdash e\ \mathsf{bindex}} \qquad \textsc{RefBind}\ \frac{}{\textbf{ref} \vdash le\ \mathsf{bindex}}$$

Requires Additional Dynamic Checking

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

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

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

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

$$\textsc{CastOkRoll}\ \frac{\tau_1 \rightsquigarrow \tau'_2[\textbf{Interface}\ T\ \tau'_2 / T]}{\tau_1 \rightsquigarrow \textbf{Interface}\ T\ \tau'_2} \qquad \textsc{CastOkUnroll}\ \frac{\tau'_1[\textbf{Interface}\ T\ \tau'_1 / T] \rightsquigarrow \tau_2}{\textbf{Interface}\ T\ \tau'_1 \rightsquigarrow \tau_2}$$

Figure 5: Typing of imperative and type casting constructs of JAVASCRIPTY. Ignore the CASTOKROLL and CASTOKUNROLL rules unless attempting the extra credit implementation.

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

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

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

$$\textsc{DoPlusString}\ \frac{str' = str_1 + str_2}{\langle M, str_1 + str_2\rangle \longrightarrow \langle M, str'\rangle}$$

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

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

$$\textsc{DoEquality}\ \frac{b' = (v_1\ bop\ v_2) \qquad bop \in \{===, !==\}}{\langle M, v_1\ bop\ v_2\rangle \longrightarrow \langle M, b'\rangle}$$

$$\textsc{DoAndTrue}\ \frac{}{\langle M, \textbf{true}\ \&\&\ e_2\rangle \longrightarrow \langle M, e_2\rangle} \qquad \textsc{DoAndFalse}\ \frac{}{\langle M, \textbf{false}\ \&\&\ e_2\rangle \longrightarrow \langle M, \textbf{false}\rangle} \qquad \textsc{DoOrTrue}\ \frac{}{\langle M, \textbf{true}\ ||\ e_2\rangle \longrightarrow \langle M, \textbf{true}\rangle}$$

$$\textsc{DoOrFalse}\ \frac{}{\langle M, \textbf{false}\ ||\ e_2\rangle \longrightarrow \langle M, e_2\rangle} \qquad \textsc{DoPrint}\ \frac{v_1 \text{ printed}}{\langle M, \textbf{console.log}(v_1)\rangle \longrightarrow \langle M, \textbf{undefined}\rangle} \qquad \textsc{DoIfTrue}\ \frac{}{\langle M, \textbf{true}\ ?\ e_2 : e_3\rangle \longrightarrow \langle M, e_2\rangle}$$

$$\textsc{DoIfFalse}\ \frac{}{\langle M, \textbf{false}\ ?\ e_2 : e_3\rangle \longrightarrow \langle M, e_3\rangle} \qquad \textsc{SearchUnary}\ \frac{\langle M, e_1\rangle \longrightarrow \langle M', e_1'\rangle}{\langle M, uop\, e_1\rangle \longrightarrow \langle M', uop\, e_1'\rangle} \qquad \textsc{SearchBinary}_1\ \frac{\langle M, e_1\rangle \longrightarrow \langle M', e_1'\rangle}{\langle M, e_1\ bop\ e_2\rangle \longrightarrow \langle M', e_1'\ bop\ e_2\rangle}$$

$$\textsc{SearchBinary}_2\ \frac{\langle M, e_2\rangle \longrightarrow \langle M', e_2'\rangle}{\langle M, v_1\ bop\ e_2\rangle \longrightarrow \langle M', v_1\ bop\ e_2'\rangle} \qquad \textsc{SearchPrint}\ \frac{\langle M, e_1\rangle \longrightarrow \langle M', e_1'\rangle}{\langle M, \textbf{console.log}(e_1)\rangle \longrightarrow \langle M', \textbf{console.log}(e_1')\rangle}$$

$$\textsc{SearchIf}\ \frac{\langle M, e_1\rangle \longrightarrow \langle M', e_1'\rangle}{\langle M, e_1\ ?\ e_2 : e_3\rangle \longrightarrow \langle M', e_1'\ ?\ e_2 : e_3\rangle}$$

Figure 6: Small-step operational semantics of non-imperative primitives of JAVASCRIPTY. The only change compared to the previous lab is the threading of the memory.

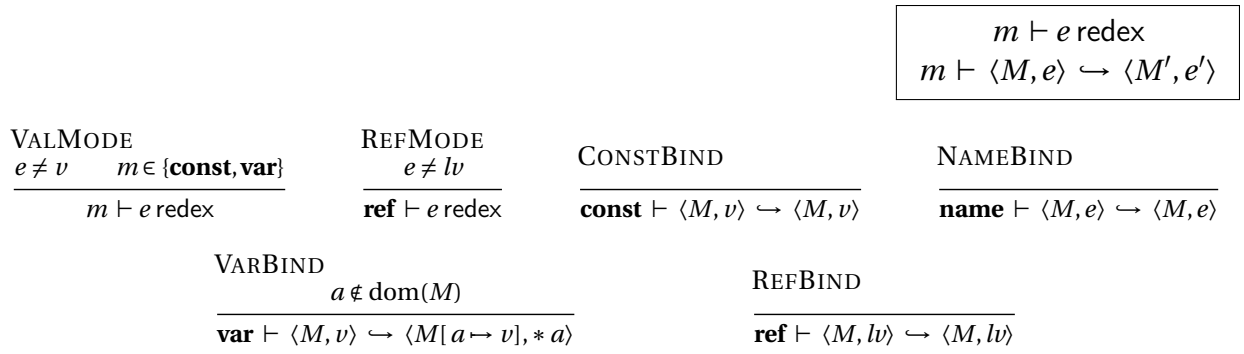

Figure 7: Define expressions that are reducible versus bindable under a mode.

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 (e.g., 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`:

```
def memalloc(k: Expr): DoWith[Mem, A]
```

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

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

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

DoDecl
$$\frac{m \vdash \langle M, e_1\rangle \hookrightarrow \langle M', e_1'\rangle}{\langle M, m\, x = e_1; e_2\rangle \longrightarrow \langle M', e_2[e_1'/x]\rangle}$$

SearchDecl
$$\frac{\langle M, e_1\rangle \longrightarrow \langle M', e_1'\rangle \qquad m \vdash e_1 \text{ redex}}{\langle M, m\, x = e_1; e_2\rangle \longrightarrow \langle M', m\, x = e_1'; e_2\rangle}$$

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

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
$$\frac{v = (x_1 : m_1\,\tau_1, \ldots, x_n : m_n\,\tau_n)\,tann => e \qquad m_i \vdash \langle M_{i+1}, e_i\rangle \hookrightarrow \langle M_i, e_i'\rangle \qquad \text{(for all } 1 \le i \le n)}{\langle M_{n+1}, v(e_1, \ldots e_n)\rangle \longrightarrow \langle M_1, e[e_n'/x_n]\cdots[e_1'/x_1]\rangle}$$

DoCallRec
$$\frac{v = x(x_1 : m_1\,\tau_1, \ldots, x_n : m_n\,\tau_n)\,tann => e \qquad m_i \vdash \langle M_{i+1}, e_i\rangle \hookrightarrow \langle M_i, e_i'\rangle \qquad \text{(for all } 1 \le i \le n)}{\langle M, v(e_1, \ldots e_n)\rangle \longrightarrow \langle M, e[e_n'/x_n]\cdots[e_1'/x_1][v/x]\rangle}$$

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$_2$
$$\frac{\langle M, e_i\rangle \longrightarrow \langle M', e_i'\rangle \qquad m_i \vdash e_i \text{ redex} \qquad m_j \nvdash e_j \text{ redex} \qquad \text{for all } j < i}{\langle M, \big(p(\overline{x : \varsigma}) => e\big)(v_1, \ldots, v_{i-1}, e_i, \ldots, e_n)\rangle \longrightarrow \langle M', \big(p(\overline{x : \varsigma}) => e\big)(v_1, \ldots, v_{i-1}, e_i', \ldots, e_n)\rangle}$$

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

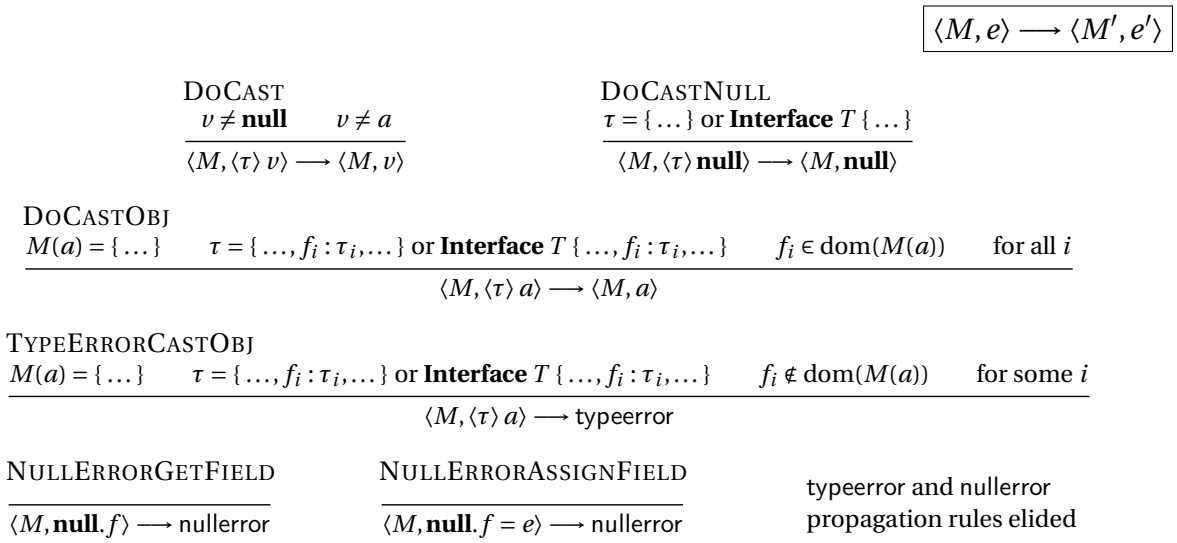

Figure 9: Small-step operational semantics of type casting and null dereference errors of JAVASCRIPTY. Ignore the "or **Interface** ..." parts unless attempting the extra credit implementation.

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 lower(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 `lower` function, we need to apply this type replacement anywhere the 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.