CSCI 3155: Lab Assignment 5

Spring 2012: Due Friday, April 19, 2012

The primary purpose of this lab to explore mutation or imperative updates in programming languages. A related topic that we also explore is parameter passing modes. Concretely, we will 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 object-orientation (i.e., 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 another 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. You will work on this assignment in pairs. However, note that **each student needs to submit a write-up** 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.

Recall the evaluation guideline from the course syllabus. 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 2.9.2). A program that does not compile will *not* be graded.

Submission Instructions. First, submit your `.scala` file to the auto-testing system to get instant feedback on your code. You may submit as many times as you want, but please submit at least once before the deadline. We will only look at your last submission.

Then, submit to the github the files mentioned in README.md

Getting Started. The code template `Lab5.scala` is in the folder /src/main/scala. Be sure to also write your name, your partner, and collaborators in the comment header.

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

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

Figure 1: Abstract Syntax of JavaScripty

2. 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 (unlike the previous lab).

Parameter Passing Modes. In this lab, we annotate function parameters with **const**, **var**, **name**, or **ref**. In the concrete syntax, we can leave off the annotation in which case we consider it as **var** (like JavaScript). The annotations **const** and **var** say the parameters should be pass-by-value (as before) either with an immutable or mutable variable, respectively. The **name** and **ref** annotations specify pass-by-name and pass-by-reference, respectively.

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
jsy.print(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 values. 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 ideally nowhere else. Our `step` implementation should only result in throwing `DynamicTypeError` when executing a cast. For simplicitly, we limit the expressivity of casts to essentially 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. For simplicity, 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.

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 τ that bundles the type name T with its definition τ. In **Interface** T τ, the type variable T should be considered bound in this construct.

- This "lowering" is implemented in the function

  def `removeInterfaceDecl(e: Expr): Expr`

  that you need to complete.

  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]`).

  Code to perform the replacement over types τ (i.e., `Typ`) is provided in the template, which recursively walks over the structure of types and applies the replacement to type variables (i.e, `TVar`) using the type variable environment.

  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. For simplicity, we assume that τ in an **interface** type declaration does not include any other type name besides T. This assumption means that something like the following code:

  interface `A { ... }`
  interface `B { a: A }`

  is unexpected input. Do not worry about checking for this case (just be aware that some later phase will probably crash).

  For **extra credit**, extend your code to handle this case. Remember that extra credit is really "extra," so only tackle this after you have everything else working. If you try the extra credit, mention it in your write-up. We will look at it if everything else is working well.

With objects allocated on the heap, we also introduce the **null** value, which enables pointer-based data structures. Somewhat uniquely, **null** 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 `Uop`s (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.

- 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 6, 7, and 8.

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(m: Mem, e: Expr): (Mem, Expr)
  ```

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

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

```
/* 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
  Const   const
case object Var extends Mutability
  Var   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(a: Int) extends Expr
  A(...)   a
case object Deref extends Uop
  Deref   *

/* Functions */
case class Function(p: Option[String],
                    params: List[(String,(PMode,Typ))], tann: Option[Typ],
                    e1: Expr) extends Expr
  Function(p, (x,(mode,τ)), tann, e1)   function p(mode x : τ)tann e1

sealed abstract class PMode
case object PConst extends PMode
  PConst   const
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.

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

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

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

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

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

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

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

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

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

$$\textsc{TypeString}\ \frac{}{\Gamma \vdash \mathit{str} : \textbf{string}} \qquad \textsc{TypeUndefined}\ \frac{}{\Gamma \vdash \textbf{undefined} : \textbf{Undefined}} \qquad \textsc{TypeObject}\ \frac{\Gamma \vdash e_i : \tau_i \qquad \text{(for all } i\text{)}}{\Gamma \vdash \{\ \ldots, f_i : e_i, \ldots\ \} : \{\ \ldots, f_i : \tau_i, \ldots\ \}}$$

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

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

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

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

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

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

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

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

Figure 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.

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 can call `A.fresh()` to get a fresh address.

- The `step` function now has the following signature

  ```
  def step(m: Mem, e: Expr): (Mem, Expr)
  ```

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

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

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

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

$$\textsc{TypeCall}\ \frac{\Gamma \vdash e : (\ mode_1\, x_1 : \tau_1, \ldots, mode_i\, x_i : \tau_i, \ldots, mode_n\, x_n : \tau_n) \Rightarrow \tau \qquad \Gamma \vdash e_i : \tau_i \qquad e_i = le_i \text{ if } mode_i = \mathbf{ref} \qquad \text{(for all i)}}{\Gamma \vdash e(e_1, \ldots, e_i, \ldots, e_n) : \tau}$$

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

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

$$\textsc{CastOkEq}\ \frac{}{\tau \rightsquigarrow \tau} \qquad \textsc{CastOkNull}\ \frac{}{\mathbf{Null} \rightsquigarrow \{\ \ldots\ \}}$$

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

$$\textsc{CastOkObject}_{\downarrow}\ \frac{\tau_i = \tau_i' \qquad \text{(for all } 1 \le i \le n \le m)}{\{\ \ldots, f_i : \tau_i, \ldots, f_n : \tau_n\ \} \rightsquigarrow \{\ \ldots, f_i : \tau_i', \ldots, f_n : \tau_n', \ldots, f_m : \tau_m'\ \}} \qquad \textsc{CastOkRoll}\ \frac{\tau_1 \rightsquigarrow \tau_2'[\mathbf{Interface}\ T\ \tau_2'/T]}{\tau_1 \rightsquigarrow \mathbf{Interface}\ T\ \tau_2'}$$

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

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

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 2.2). 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, the `substitute` function is provided completely, but it calls the function

  def `avoidCapture(esub: Expr, e: Expr): Expr`

  to rename bound variables in `e` to avoid capturing free variables in `esub`. It works in a similar way to `substitute` and `removeInterfaceDecl` that do some transformation with with variables in some way.

  First, we figure out the free variables of `esub`. Then, we perform a recursive walk over expressions with an environment mapping original names to new non-clashing names with the free variables of `esub` (i.e., `Map[String, String]`. We need to pick a non-clashing renaming at places where variables around (i.e., `Decl` and `Function`) and then rename uses of those variables where they are in scope.

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 semantidefine (if the challenge excites you!).

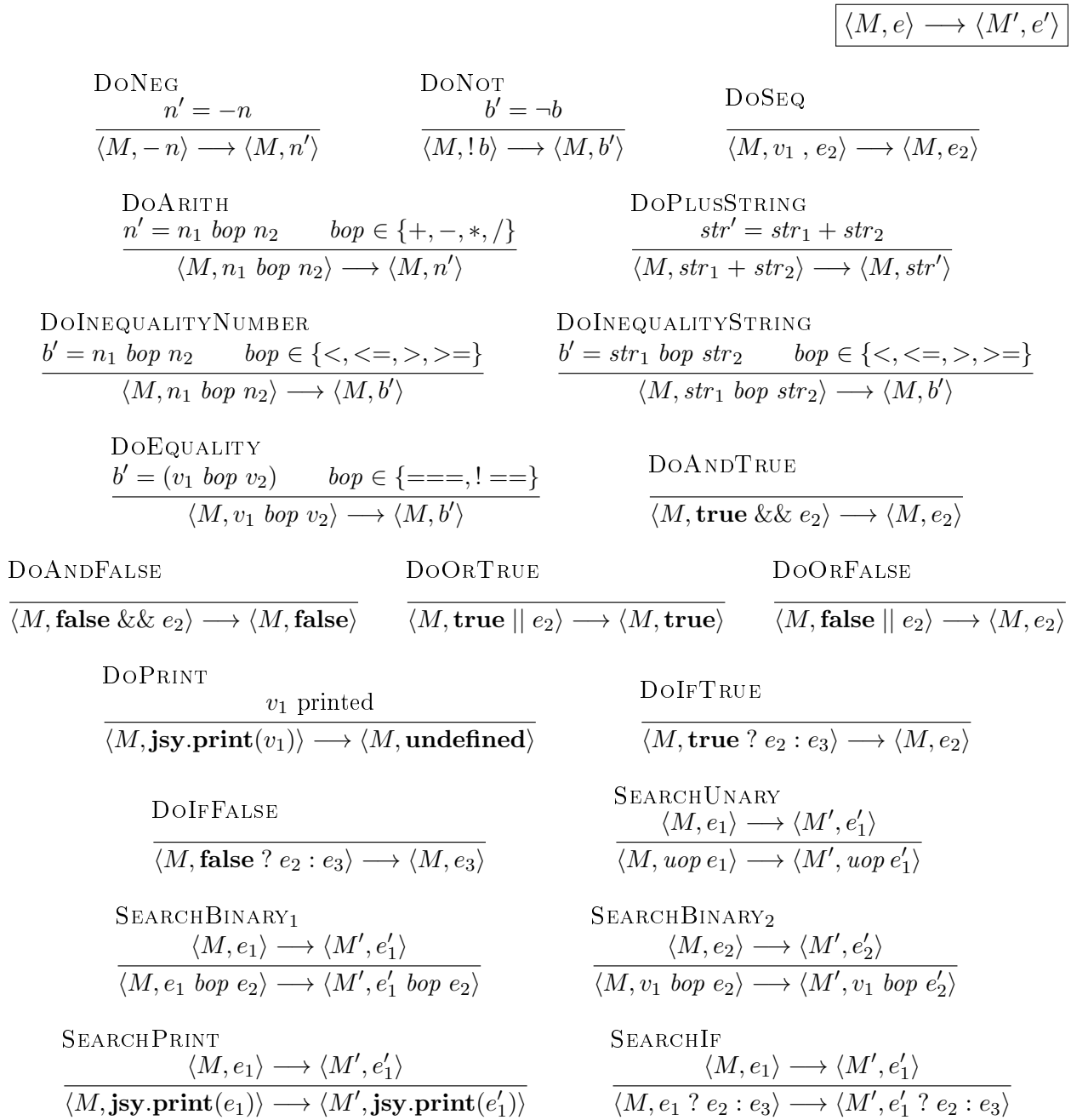

Figure 6: Small-step operational semantics of non-imperative primitives of JavaScripty. The only change compared to the previous lab is the threading of the memory.

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

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

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

SearchAssign$_2$
$$\frac{\langle M, e_2\rangle \longrightarrow \langle M', e_2'\rangle}{\langle M, \mathit{lv}_1 = e_2\rangle \longrightarrow \langle M', \mathit{lv}_1 = e_2'\rangle}$$

DoCall
$$\frac{\begin{array}{l} v = \mathbf{function}\ (\ \mathit{mode}_1\, x_1 : \tau_1, \ldots, \mathit{mode}_n\, x_n : \tau_n)\mathit{tann}\ e \qquad e_i = v_i \text{ if } \mathit{mode}_i \in \{\mathbf{const}, \mathbf{var}\} \\ \qquad e_i = \mathit{lv}_i \text{ if } \mathit{mode}_i \in \{\mathbf{ref}\} \qquad e_i' = e_i \text{ if } \mathit{mode}_i \in \{\mathbf{const}, \mathbf{name}, \mathbf{ref}\} \\ e_j' = {*}\, a_j \qquad a_j \notin \mathrm{dom}(M) \qquad a_j \neq a_k \qquad (\text{for all } i, \text{for all } j \text{ where } \mathit{mode}_j = \mathbf{var}, \text{for all } k \neq j) \end{array}}{\langle M, v(e_1, \ldots e_n)\rangle \longrightarrow \langle M\overline{[\, a_j \mapsto v_j]}, e[e_n'/x_n] \cdots [e_1'/x_1]\rangle}$$

DoCallRec
$$\frac{\begin{array}{l} v = \mathbf{function}\ (\ \mathit{mode}_1\, x_1 : \tau_1, \ldots, \mathit{mode}_n\, x_n : \tau_n)\mathit{tann}\ e \qquad e_i = v_i \text{ if } \mathit{mode}_i \in \{\mathbf{const}, \mathbf{var}\} \\ \qquad e_i = \mathit{lv}_i \text{ if } \mathit{mode}_i \in \{\mathbf{ref}\} \qquad e_i' = e_i \text{ if } \mathit{mode}_i \in \{\mathbf{const}, \mathbf{name}, \mathbf{ref}\} \\ e_j' = {*}\, a_j \qquad a_j \notin \mathrm{dom}(M) \qquad a_j \neq a_k \qquad (\text{for all } i, \text{for all } j \text{ where } \mathit{mode}_j = \mathbf{var}, \text{for all } k \neq j) \end{array}}{\langle M, v(e_1, \ldots e_n)\rangle \longrightarrow \langle M\overline{[\, a_j \mapsto v_j]}, e[e_n'/x_n] \cdots [e_1'/x_1][v/x]\rangle}$$

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$_2$
$$\frac{\begin{array}{l} v = \mathbf{function}\ (\ \mathit{mode}_1\, x_1 : \tau_1, \ldots, \mathit{mode}_n\, x_n : \tau_n)\mathit{tann}\ e \qquad \langle M, e_i\rangle \longrightarrow \langle M', e_i'\rangle \\ e_j = v_j \text{ if } \mathit{mode}_i \in \{\mathbf{const}, \mathbf{var}\} \qquad e_j = \mathit{lv}_j \text{ if } \mathit{mode}_i \in \{\mathbf{ref}\} \qquad (\text{for all } j < i) \end{array}}{\langle M, v(e_1, \ldots, e_i, \ldots, e_n)\rangle \longrightarrow \langle M', v(e_1, \ldots, e_i', \ldots, e_n)\rangle}$$

Figure 7: Small-step operational semantics of objects, binding constructs, variable and field assignment, and function call of JavaScripty.

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

$$\textsc{DoCast}\ \frac{v \neq \textbf{null} \qquad v \neq a}{\langle M, \langle \tau \rangle\, v\rangle \longrightarrow \langle M, v\rangle} \qquad \textsc{DoCastNull}\ \frac{\tau = \{\ \ldots\ \} \text{ or } \textbf{Interface}\ T\ \{\ \ldots\ \}}{\langle M, \langle \tau \rangle\, \textbf{null}\rangle \longrightarrow \langle M, \textbf{null}\rangle}$$

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

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

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

$\mathsf{typeerror}$ and $\mathsf{nullerror}$ propagation rules elided

Figure 8: Small-step operational semantics of type casting and null dereference errors of JavaScripty.