Homework 6

Due date: Friday, May 15, 2015

The primary purpose of this assignment is to explore subtyping in object-oriented programming languages. We explore this language feature together with recursive types. Concretely, we extend JakartaScript with subtyping, interfaces, and casts. These new language extensions will significantly increase the practical expressiveness of our language compared to our previous interpreters. Our interpreter will now support many of the core features of modern programming languages.

Like last time, 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 exercises in larger groups. However, we ask that you write up your answers in pairs. Also, be sure to acknowledge those with which you discussed, including your partner and those outside of your pair.

Try to make your code as concise and clear as possible. Challenge yourself to find the most crisp, concise way of expressing the intended computation. This may mean using ways of expressing computation currently unfamiliar to you.

Finally, make sure that your file compiles and runs (using Scala 2.11.5). A program that does not compile will *not* be graded.

Submission instructions. Upload to NYU classes exactly three files named as follows:

- `HW6-YourNYULogin.pdf` with your answers to the written questions (**only needed for the bonus question**)
- `HW6-YourNYULogin.scala` with your answers to the coding exercises.
- `HW6Spec-YourNYULogin.scala` with any updates to your unit tests.
- `HW6-YourNYULogin.js` with a challenging test case for your JakartaScript interpreter.

Replace `YourNYULogin` with your NYU login ID (e.g., I would submit `HW6-tw47.pdf` and so forth). To help with managing the submissions, we ask that you rename your uploaded files in this manner.

Getting started. Download the code pack `hw6.zip` from the assignment section on the NYU classes page.

Problem 1 JakartaScript Interpreter with Subtyping (64 Points)

The syntax of the version of JakartaScript that we are going to implement in this assignment is shown in Figure 1 with the new extensions highlighted. We discuss the new features in more detail.

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

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

This interface declaration says: declare a type name T defined to be type τ where the scope of T is the type τ and the expression e. Since τ may refer to T, we can use interface declarations to define recursive types. This feature is useful to implement recursive data structures such as lists and trees.

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. So, instead, we replace T with a new type of the form **Interface** $T\ \tau$ that bundles the type name T with its definition τ. In **Interface** $T\ \tau$, the type variable T should be considered bound in this construct. This "lowering" is implemented in the function

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

that is provided for you. This function is very similar to substitution, but instead of substituting for program variables x (i.e., `Var(x)`), we substitute for type variables T (i.e., `TVar(T)`). Thus, we need an environment that maps type variable names T to types τ (i.e., an `env` parameter of type `Map[String,Typ]`). In the `removeInterfaceDecl` function, we need to apply this type replacement anywhere the JakartaScript 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 **interface** $T\ \tau\texttt{;}\ e$ by extending the environment with $[T \mapsto \textbf{Interface}\ T\ \tau]$ and applying the replacement in e.

Casts and Null Pointers. In the previous homework assignment, 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 where we 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 $<\tau>\ e$. Executing a cast may result in a dynamic type error when the cast is evaluated and e's dynamic type is not a subtype of τ. For simplicity, we will not actually check the dynamic type of e against τ when we evaluate a cast. Instead, a cast $<\tau>\ e$ simply evaluates to e. We will then throw a `DynamicTypeError` whenever our interpreter gets stuck in an expression that is not a value and cannot be further reduced (essentially replacing the `SuckError`

exception in our previous interpreter). A correct implementation of the interpreter should only ever throw a `DynamicTypeError` exception when it has previously executed a bad cast.

With objects allocated on the heap, we also introduce the **null** value, which enables pointer-based data structures. The **null** value has type **Null**, which we make a subtype of all object types. That is, we can use **null** in any context that expects an object. 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`.

Parameter Passing Modes. We simplify parameter passing compared to Homework 5 by only considering one passing mode. More precisely, we treat all parameters of functions as **const** parameters, as we did in Homework 3 and 4. We do this for pedagogical reasons so that you do not have to think about the implications of parameter passing modes when you implement subtyping.

In Figure 2, we show the updated and new AST nodes. Note that `Deref` is a `Uop` and `Assign` a `Bop`.

Typing Relations. The inference rules defining the typing relation are given in Figures 3 and 4. Figure 4 summarizes the rules of the new primitives and the rules that involve subtyping. Note that we only allow up- and down-casts. The subtyping relation itself is defined in Figure 5. It is formalized by judgments of the form

$$C \vdash \tau <: \tau'$$

which say that under coinduction hypotheses C, τ is a subtype of τ'. The set C collects the subtyping constraints involving interface types that we have seen so far along the trail of the derivation. We can use these constraints as hypotheses to prove repeated occurrences of the same constraints during the recursive traversal of the type expressions. Keeping track of these hypotheses ensures termination of the derivations that involve recursive interface types. The judgment form $\tau <: \tau'$ that we use in the rules of the typing relation in Figure 3 is simply a short-hand for $\emptyset \vdash \tau <: \tau'$.

Joins and Meets. The typing rule TypeIf for typing conditional expressions uses the judgment $\tau_2 \sqcup \tau_3 = \tau$ to compute the *join* (i.e., the least common supertype) of the types τ_2 and τ_3. Similar to the subtyping relation, we formalize the computation of joins by judgments of the form $C \vdash \tau_2 \sqcup \tau_3 = \tau$ where C keeps track of the joins of interface types that we encountered along the current derivation trail. Since function types are contravariant in their parameter types, computing the joins of such types involves the computation of meets (i.e., greatest common subtypes). We therefore have to compute joins and meets in parallel. The algorithm to compute meets of two types τ_1 and τ_2 is formalized by the judgment form $C \vdash \tau_1 \sqcap \tau_2 = \tau$. The rules for joins are summarized in Figure 6 and the rules for meets are given in Figure 7. In the rules for object types we use the notation

$$\{\, f : \tau \mid P(f, \tau) \,\}$$

to denote the object type that consists of all fields f with types τ that satisfy the condition $P(f, \tau)$. Moreover, for an object type τ we write $(f, \tau') \in \tau$ to say that f is a field of τ and f has type τ'. Finally, for two object types τ_1 and τ_2, we denote by $\tau_1 \uplus \tau_2$ the disjoint union of the two object types. That is, $\tau_1 \uplus \tau_2$ is the object type that comprises all the fields of τ_1 and τ_2 with their respective types that are not common to both τ_1 and τ_2. For example

$$\{f : \mathbf{number}; g : \mathbf{bool}; h : \{\}\} \uplus \{g : \mathbf{number}; k : \mathbf{bool}; f : \mathbf{number}\} = \{h : \{\}; k : \mathbf{bool}\}$$

In the case where the join of two types τ_1 and τ_2 does not exist, we write $C \vdash \tau_1 \sqcup \tau_2 = \bot$ to indicate the failure of the join computation (and similarly for meets). Note that the rule MeetObjNull is the only rule that recovers from failure in a subderivation. In all other cases, failure is simply propagated. We elide the failure propagation rules for brevity.

Implementation of Type Inference. As in Homework 5, we implement type inference with the function

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

which you need to complete. In your Scala code, you can write `t1 <:< t2` to check whether `t1` is a subtype of `t2`, and `t1 |:| t2` to compute the join of two types. The join operator returns a value of type `Option[Typ]` where `None` indicates that the join of the arguments does not exist. These operators are defined in the class `Subtyp` and implemented as calls to other functions that you also need to complete. These other functions are described below.

Matching Modulo Unfolding of Interface Types. Note that in the rules of Figures 3 and 4, we consider types to be equal up to unfolding. That is, we think of interface types as the (possibly infinite) type expression trees that we obtain by unfolding them completely. With this semantics of interface types, the following property holds, which states that an interface type is equal to its own unfolding:

$$\mathbf{Interface}\ T\ \tau = \tau[(\mathbf{Interface}\ T\ \tau)/T]$$

For example, this means that the following equalities hold

$$\mathbf{number} = (\mathbf{Interface}\ T_1\ \mathbf{number}) = (\mathbf{Interface}\ T_1\ (\mathbf{Interface}\ T_2\ \mathbf{number}) = \ldots$$

Consequently, when you implement the typing rules and you pattern match on the constructors of type expressions, you have to take unfolding of interface types into account. For instance, the following implementation of the rule TypeUMinus that we used in the previous homework assignments will no longer work correctly:

```
case UnOp(UMinus, e1) => typ(e1) match {
   case TNumber => TNumber
   case tgot => err(tgot, e1)
}
```

The problem is that the first **case** clause should also match types of the form

```
TInterface(T, TNumber)
```

and so forth. To deal with this problem gracefully, we use Scala's `unapply` mechanism for defining custom patterns. Specifically, we have defined a pattern constructor `TUnfold` that implicitly unfolds a type expression to the point where the top-level type expression constructor is not `TInterface`. This allows you to write simple patterns that match modulo unfolding of interface types. You can use this custom pattern by wrapping the actual pattern that you want to match against in the `TUnfold` constructor. For example, the rule for TypeUMinus will now look as follows:

```
case UnOp(UMinus, e1) => typ(e1) match {
   case TUnfold(TNumber) => TNumber
   case tgot => err(tgot, e1)
}
```

The pattern `TUnfold(TNumber)` will match the type `TNumber`, as well as all interface types that are equal to `TNumber` modulo unfolding, such as `TInterface(T, TNumber)`, etc. Keep in mind that when you write patterns that match the types that you infer for subexpressions in `typeInfer`, you always have to wrap those patterns in `TUnfold`. The function `typeInfer` is the only place where you need to do this. Everywhere else, we have to deal with interface types explicitly.

Implementation of Subtyping. The actual subtyping relation is implemented by the function

```
def subtype(s: Typ, t: Typ): StateBoolean[Set[(Typ, Typ)]]
```

The definition of the subtyping relation in Figure 5 is suboptimal. In fact, it has worst-case exponential time complexity. The exponential explosion can be avoided by threading the hypothesis set C through the entire derivation, instead of recomputing it for each subderivation. We realize this threading using the class `StateBoolean[S]`. This class extends the state monad `State[S,Boolean]` from the previous homework, i.e., it represents a computation over an input/output state of type `S` that produces a Boolean result value. We instantiate `StateBoolean` with the type `Set[(Typ,Typ)]`, which we use to represent the hypothesis set C.

In addition to the methods that are provided by class `State`, the class `StateBoolean` has additional methods for combining `StateBoolean` values using Boolean connectives. For example, suppose that `P` and `Q` are both values of type `StateBoolean[S]`. Then `P && Q` yields a new value of type `StateBoolean[S]` that encapsulates a computation which returns **`true`** if both `P` and `Q` return **`true`**. As for standard Boolean connectives, the operator `&&` on `StateBoolean[S]` is short-circuiting. Similarly, `StateBoolean[S]` provides an operator `||` to compute the "or" of the results of two stateful Boolean computations. You can use these operators to easily combine the results of recursive calls to `subtype`.

The function `subtype` is already provided for you. It implements the basic plumbing for maintaining and checking the hypothesis set C (essentially the rule SubCoind and the updates of C for $\text{SubInterface}_{1,2}$). The remaining rules are handled by the function

```
def subtypeBasic(s: Typ, t: Typ): StateBoolean[Set[(Typ, Typ)]]
```

which is already partially implemented. You need to complete the missing (interesting) cases, which are the rules SUBFUN and SUBOBJ. Note that all recursive subtyping checks that are needed to implement the premises of these rules must go to the function `subtype` instead of `subtypeBasic`. Otherwise, the derivations may not terminate for recursive types.

The computation of joins and meet is realized by the functions

```
def join(s: Typ, t: Typ): StateOption[Cache, Typ]
```

respectively

```
def meet(s: Typ, t: Typ): StateOption[Cache, Typ]
```

Again, we use a state monad to thread the hypothesis set C through the computation. We use the special state monad `StateOption[S,R]` which extends `State[S,Option[R]]`. That is `StateOption[S,R]` encapsulates a computation over an input/output state of type `S` that returns a value of type `Option[Typ]`. The class `StateOption[S,R]` lifts some useful methods, such as `orElse`, from `Option` to `StateOption`.

The type `Cache` is an alias for the type `Map[(Typ,Bound,Typ),Typ]`, which is our representation of the hypothesis set C for the join and meet computation.

Again, the functions `join` and `meet` do the plumbing for the rules that involve look-up and update of the hypothesis set for dealing with recursive interface types. The remaining rules are handled by the functions:

```
def joinBasic(s: Typ, t: Typ): StateOption[Cache, Typ]
```

respectively

```
def meetBasic(s: Typ, t: Typ): StateOption[Cache, Typ]
```

Complete the missing cases in these functions.

To understand the join and meet calculation better, we suggest that you manually calculate the joins and meets of some object types, before you start with the implementation of these functions. In particular, you should also consider object types that involve recursive interface types in this exercise. Once you have implemented your interpreter, you can validate the manually calculated joins and meets against the results produced by your implementation. To do so, write small test programs whose inferred types correspond to the joins and meets of the types you considered. See, e.g., the programs `joinSimple.js` and `meetSimple.js` that are provided as benchmarks in the code package. If you run your interpreter with the option `-d -ne`, then it will only type check the input program and print the inferred type without actually evaluating the program. Alternatively, you can also write unit tests that exercise your implementation of the join and meet functions. Some unit tests are already provided for you.

Reduction. The new small-step operational semantics is given in Figures 8, 9, and 10. There is little change compared to Homework 6. The new or modified rules are summarized in Figure 10. The step relation is again implemented by the function

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

Most of the function is provided for you. You only need to add the new rules of Figure 10 and complete the rules for function calls.

Problem 2 Implementing Runtime Checks for Casts (14 Bonus Points)

Our operational semantics of cast expressions is not ideal because we ignore them and delay failure on a bad cast to the point where the `step` relation gets stuck in the evaluation. A better approach to handle cast expressions $<\tau> e$ is to first evaluate e to a value v and then check whether the runtime type of v is indeed a subtype of τ before we eliminate the cast. If the check fails, we immediately abort evaluation with a dynamic type error. How could this dynamic type check be implemented, what difficulties would we encounter, and how could we solve these difficulties? First, give the explanation in prose, and then, try to formalize it in our semantics and type system (if the challenge excites you!). You do not have to provide all the required additional rules, but only some interesting ones.

$n \in Num$		numbers (double)
$s \in Str$		strings
$a \in Addr$		addresses
$b \in Bool$	$::= \textbf{true} \mid \textbf{false}$	Booleans
$x \in Var$		variables
$f \in Fld$		field names
$T \in TVar$		type variables
$\tau \in Typ$	$::= \textbf{bool} \mid \textbf{number} \mid \textbf{string} \mid \textbf{Undefined} \mid \textbf{Null} \mid$	types
	$(\overline{x : \tau}) \Rightarrow \tau_0 \mid \{f_1 : \tau_1 ; \ldots ; f_n : \tau_n\} \mid$	
	$T \mid \textbf{Interface}\ T\ \tau$	
$v \in Val$	$::= \textbf{undefined} \mid n \mid b \mid s \mid a \mid \textbf{null} \mid \mathsf{nullerror} \mid$	values
	$\textbf{function}\ p\ (\overline{x : \tau})\ t\ e$	
$e \in Expr$	$::= x \mid v \mid uop\ e \mid e_1\ bop\ e_2 \mid e_1\ ?\ e_2\ :\ e_3 \mid$	expressions
	$\textbf{console.log}(e) \mid e_1\ (\overline{e}) \mid e\,.\,f \mid \{\overline{f : e}\} \mid$	
	$mut\ x = e_1 ; e_2$	
$lv \in LVal$	$::= \star\, a \mid a\,.\,f$	location values
$le \in LExpr$	$::= x \mid e\,.\,f$	location expressions
$uop \in Uop$	$::= - \mid ! \mid \star \mid \texttt{<}\tau\texttt{>}$	unary operators
$bop \in Bop$	$::= + \mid - \mid \star \mid / \mid \texttt{===} \mid \texttt{!==} \mid < \mid > \mid$	binary operators
	$\texttt{<=} \mid \texttt{>=} \mid \texttt{\&\&} \mid \texttt{||} \mid \texttt{,} \mid \texttt{|=}$	
p	$::= x \mid \epsilon$	function names
t	$::= : \tau \mid \epsilon$	return types
$mut \in Mut$	$::= \textbf{const} \mid \textbf{var}$	mutability
$k \in Con$	$::= v \mid \{\overline{f : v}\}$	memory contents
$M \in Mem$	$= Addr \rightharpoonup Con$	memories

Figure 1: Abstract syntax of JakartaScript

```
sealed abstract class Expr extends Positional
...

/** Functions */
type Params = List[(String,Typ)]
case class Function(p: Option[String], xs: Params, t: Option[Typ], e1: Expr) extends Expr
Function(p, x : τ, t, e1) function p (x : τ) t e1

/** Addresses and Mutation */
...
case object Null extends Expr
Null null

/** Casts */
case class Cast(t: Typ) extends Uop
Cast(τ) <τ>

/** Types */
sealed abstract class Typ
...
case 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 JAKARTASCRIPT. After each **`case class`** or **`case object`**, we show the correspondence between the representation and the concrete syntax.

$$\frac{}{\Gamma \vdash b : \textbf{bool}}\ \textsc{TypeBool} \qquad \frac{}{\Gamma \vdash n : \textbf{number}}\ \textsc{TypeNum} \qquad \frac{}{\Gamma \vdash s : \textbf{string}}\ \textsc{TypeStr}$$

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

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1\,\texttt{,}\ e_2 : \tau_2}\ \textsc{TypeSeq} \qquad \frac{\Gamma \vdash e_1 : \textbf{bool} \quad \Gamma \vdash e_2 : \textbf{bool} \quad \mathit{bop} \in \{\texttt{\&\&}, \texttt{||}\}}{\Gamma \vdash e_1\,\mathit{bop}\,e_2 : \textbf{bool}}\ \textsc{TypeAndOr}$$

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash \textbf{console.log}\,(e)\ : \textbf{Undefined}}\ \textsc{TypePrint}$$

$$\frac{\Gamma \vdash e : \textbf{number}}{\Gamma \vdash -e : \textbf{number}}\ \textsc{TypeUMinus} \qquad \frac{\Gamma \vdash e : \textbf{bool}}{\Gamma \vdash\ !\ e : \textbf{bool}}\ \textsc{TypeNot}$$

$$\frac{\Gamma \vdash e_1 : \textbf{string} \quad \Gamma \vdash e_2 : \textbf{string}}{\Gamma \vdash e_1 + e_2 : \textbf{string}}\ \textsc{TypePlusStr}$$

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

$$\frac{\Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \tau \quad \tau \in \{\textbf{number}, \textbf{string}\} \quad \mathit{bop} \in \{>, >=, <, <=\}}{\Gamma \vdash e_1\ \mathit{bop}\ e_2 : \textbf{bool}}\ \textsc{TypeInequal}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \ldots \quad \Gamma \vdash e_n : \tau_n}{\Gamma \vdash \{\,f_1\texttt{:}e_1\texttt{,}\ \ldots\texttt{,}\ f_n\texttt{:}e_n\,\} : \{\,f_1\texttt{:}\tau_1\texttt{;}\ \ldots\texttt{;}\ f_n\texttt{:}\tau_n\,\}}\ \textsc{TypeObj}$$

$$\frac{\Gamma \vdash e : \{\,f_1\texttt{:}\tau_1\texttt{;}\ \ldots\texttt{;}\ f_n\texttt{:}\tau_n\,\} \quad f = f_i \quad \tau = \tau_i \quad i \in \{1, \ldots, n\}}{\Gamma \vdash e\texttt{.}f : \tau}\ \textsc{TypeGetField}$$

$$\frac{\Gamma \vdash e_d : \tau_d \quad \Gamma' = \Gamma[x \mapsto (\mathit{mut}, \tau_d)] \quad \Gamma' \vdash e_b : \tau_b}{\Gamma \vdash \mathit{mut}\ x = e_d\texttt{;}\ e_b : \tau_b}\ \textsc{TypeDecl}$$

$$\frac{x \in \mathit{dom}(\Gamma) \quad \Gamma(x) = (\mathit{mut}, \tau)}{\Gamma \vdash x : \tau}\ \textsc{TypeVar}$$

$$\frac{\begin{array}{l}\Gamma' = \Gamma[x_1 \mapsto (\textbf{const}, \tau_1)] \ldots [x_n \mapsto (\textbf{const}, \tau_n)] \\ \Gamma' \vdash e : \tau \quad \tau' = (x_1\texttt{:}\tau_1\texttt{,}\ \ldots\texttt{,}\ x_n\texttt{:}\tau_n) \Rightarrow \tau\end{array}}{\Gamma \vdash \textbf{function}\,(x_1\texttt{:}\tau_1\texttt{,}\ \ldots\texttt{,}\ x_n\texttt{:}\tau_n)\,e : \tau'}\ \textsc{TypeFun}$$

Figure 3: Type checking rules for old primitives of JakartaScript that do not involve subtyping (no changes compared to Homework 5)

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2 \quad \tau_1 \sqcup \tau_2 = \tau \quad \tau_1 \text{ has no function types} \quad \mathit{bop} \in \{\texttt{===}, \texttt{!==}\}}{\Gamma \vdash e_1\ \mathit{bop}\ e_2 : \mathbf{bool}}\ \textsc{TypeEqual}$$

$$\frac{\Gamma \vdash e_1 : \tau' \quad (\tau <: \tau' \text{ or } \tau' <: \tau)}{\Gamma \vdash \texttt{<}\tau\texttt{>}\, e : \tau}\ \textsc{TypeCast}$$

$$\frac{\Gamma(x) = (\mathbf{var}, \tau') \quad \Gamma \vdash e : \tau \quad \tau <: \tau'}{\Gamma \vdash x = e : \tau}\ \textsc{TypeAssignVar}$$

$$\frac{\Gamma \vdash e_1 : \{\ \ldots; f{:}\tau'; \ldots\ \} \quad \Gamma \vdash e_2 : \tau \quad \tau <: \tau'}{\Gamma \vdash e_1\texttt{.}f = e_2 : \tau}\ \textsc{TypeAssignFld}$$

$$\frac{\Gamma \vdash e_1 : \mathbf{bool} \quad \Gamma \vdash e_2 : \tau_2 \quad \Gamma \vdash e_3 : \tau_3 \quad \tau_2 \sqcup \tau_3 = \tau}{\Gamma \vdash e_1\ \texttt{?}\ e_2\ \texttt{:}\ e_3 : \tau}\ \textsc{TypeIf}$$

$$\frac{}{\Gamma \vdash \mathbf{null} : \mathbf{Null}}\ \textsc{TypeNull} \qquad \frac{\Gamma \vdash e : (x_1\texttt{:}\tau_1\texttt{, }\ldots\texttt{, }x_n\texttt{:}\tau_n) \Rightarrow \tau \quad \text{for all } i\text{: } \Gamma \vdash e_i : \tau_i' \quad \tau_i' <: \tau_i}{\Gamma \vdash e\,(e_1, \ldots, e_n) : \tau}\ \textsc{TypeCall}$$

$$\frac{\Gamma' = \Gamma[x_1 \mapsto (\mathbf{const}, \tau_1)] \ldots [x_n \mapsto (\mathbf{const}, \tau_n)] \quad \Gamma' \vdash e : \tau_0' \quad \tau_0' <: \tau_0 \quad \tau = (x_1\texttt{:}\tau_1\texttt{, }\ldots\texttt{, }x_n\texttt{:}\tau_n) \Rightarrow \tau_0}{\Gamma \vdash \mathbf{function}\,(x_1\texttt{:}\tau_1\texttt{, }\ldots\texttt{, }x_n\texttt{:}\tau_n)\texttt{:}\tau_0\ e : \tau}\ \textsc{TypeFunAnn}$$

$$\frac{\Gamma' = \Gamma[x \mapsto \tau][x_1 \mapsto (\mathbf{const}, \tau_1)] \ldots [x_n \mapsto (\mathbf{const}, \tau_n)] \quad \Gamma' \vdash e : \tau_0' \quad \tau_0' <: \tau_0 \quad \tau = (x_1\texttt{:}\tau_1\texttt{, }\ldots\texttt{, }x_n\texttt{:}\tau_n) \Rightarrow \tau_0}{\Gamma \vdash \mathbf{function}\ x\,(x_1\texttt{:}\tau_1\texttt{, }\ldots\texttt{, }x_n\texttt{:}\tau_n)\texttt{:}\tau_0\ e : \tau}\ \textsc{TypeFunRec}$$

Figure 4: Type checking rules for new primitives of JAKARTASCRIPT and the primitives that involve subtyping ($\tau <: \tau'$ stands for $\emptyset \vdash \tau <: \tau'$ and $\tau_1 \sqcup \tau_2 = \tau_3$ stands for $\emptyset \vdash \tau_1 \sqcup \tau_2 = \tau_3$)

$$\frac{}{C \vdash \tau <: \tau}\ \textsc{SubRefl} \qquad \frac{(\tau <: \tau') \in C}{C \vdash \tau <: \tau'}\ \textsc{SubCoind} \qquad \frac{}{C \vdash \mathbf{Null} <: \{\overline{f : \tau'}\}}\ \textsc{SubNull}$$

$$\frac{C \cup \{(\mathbf{Interface}\ T\ \tau) <: \tau'\} \vdash \tau[(\mathbf{Interface}\ T\ \tau)/T] <: \tau'}{C \vdash (\mathbf{Interface}\ T\ \tau) <: \tau'}\ \textsc{SubInterface}_1$$

$$\frac{C \cup \{\tau <: (\mathbf{Interface}\ T'\ \tau')\} \vdash \tau <: \tau'[(\mathbf{Interface}\ T'\ \tau')/T']}{C \vdash \tau <: (\mathbf{Interface}\ T'\ \tau')}\ \textsc{SubInterface}_2$$

$$\frac{C \vdash \tau <: \tau' \qquad \text{for all } i\text{: } C \vdash \tau'_i <: \tau_i}{C \vdash (\,(x_1 : \tau_1, \ldots, x_n : \tau_n)\ \Rightarrow \tau) <: (\,(y_1 : \tau'_1, \ldots, y_n : \tau'_n)\ \Rightarrow \tau')}\ \textsc{SubFun}$$

$$\frac{\{g_1, \ldots, g_m\} \subseteq \{f_1, \ldots, f_n\} \quad \text{for all } i, j, \text{ if } f_i = g_j,\ C \vdash \tau_i <: \tau'_j}{C \vdash \{f_1 : \tau_1; \ldots; f_n : \tau_n\} <: \{g_1 : \tau'_1; \ldots; g_m : \tau'_m\}}\ \textsc{SubObj}$$

Figure 5: Subtyping rules for JakartaScript

$$\frac{}{C \vdash \tau \sqcup \tau = \tau}\ \textsc{JoinEq} \qquad \frac{(\tau_1 \sqcup \tau_2 = \tau) \in C}{C \vdash \tau_1 \sqcup \tau_2 = \tau}\ \textsc{JoinCoind}$$

$$\frac{C \cup \{(\mathbf{Interface}\ T_1\ \tau_1) \sqcup \tau_2 = T\} \vdash \tau_1[(\mathbf{Interface}\ T_1\ \tau_1)/T_1] \sqcup \tau_2 = \tau}{C \vdash (\mathbf{Interface}\ T_1\ \tau_1) \sqcup \tau_2 = (\mathbf{Interface}\ T\ \tau)}\ \textsc{JoinInterface}_1$$

$$\frac{C \cup \{\tau_1 \sqcup (\mathbf{Interface}\ T_2\ \tau_2) = T\} \vdash \tau_1 \sqcup \tau_2[(\mathbf{Interface}\ T_2\ \tau_2)/T_2] = \tau}{C \vdash \tau_1 \sqcup (\mathbf{Interface}\ T_2\ \tau_2) = (\mathbf{Interface}\ T\ \tau)}\ \textsc{JoinInterface}_2$$

$$\frac{C \vdash \tau_0 \sqcup \tau'_0 = \tau''_0 \quad \text{for all } i\text{: } C \vdash \tau_i \sqcap \tau'_i = \tau''_i \quad \tau'' = (\,x_1 : \tau''_1, \ldots, x_n : \tau''_n)\ \Rightarrow \tau''_0}{C \vdash (\,(x_1 : \tau_1, \ldots, x_n : \tau_n)\ \Rightarrow \tau_0) \sqcup (\,(y_1 : \tau'_1, \ldots, y_n : \tau'_n)\ \Rightarrow \tau'_0) = \tau''}\ \textsc{JoinFun}$$

$$\frac{}{C \vdash \mathbf{Null} \sqcup \{\overline{g : \tau_2}\} = \{\overline{g : \tau_2}\}}\ \textsc{JoinNull}_1 \qquad \frac{}{C \vdash \{\overline{f : \tau_1}\} \sqcup \mathbf{Null} = \{\overline{f : \tau_1}\}}\ \textsc{JoinNull}_2$$

$$\frac{\tau = \{h : \tau' \mid \exists \tau_1, \tau_2 : (h : \tau_1) \in \{\overline{f : \tau_1}\}, (h : \tau_2) \in \{\overline{g : \tau_2}\},\ \text{and } C \vdash \tau_1 \sqcup \tau_2 = \tau'\}}{C \vdash \{\overline{f : \tau_1}\} \sqcup \{\overline{g : \tau_2}\} = \tau}\ \textsc{JoinObj}$$

$$\frac{\tau_1 \neq \tau_2 \quad \tau_1 \in \{\mathbf{bool}, \mathbf{number}, \mathbf{string}, \mathbf{Undefined}\} \quad \tau_2 \neq (\mathbf{Interface}\ T\ \tau)}{C \vdash \tau_1 \sqcup \tau_2 = \bot}\ \textsc{JoinFail}_1$$

$$\frac{\tau_1 \neq \tau_2 \quad \tau_2 \in \{\mathbf{bool}, \mathbf{number}, \mathbf{string}, \mathbf{Undefined}\} \quad \tau_1 \neq (\mathbf{Interface}\ T\ \tau)}{C \vdash \tau_1 \sqcup \tau_2 = \bot}\ \textsc{JoinFail}_2$$

Figure 6: Rules for computing joins. The rules for propagating failure $\bot$ have been elided

$$\frac{}{C \vdash \tau \sqcap \tau = \tau}\ \textsc{MeetEq} \qquad \frac{(\tau_1 \sqcap \tau_2 = \tau) \in C}{C \vdash \tau_1 \sqcap \tau_2 = \tau}\ \textsc{MeetCoind}$$

$$\frac{C \cup \{(\textbf{Interface}\ T_1\ \tau_1) \sqcap \tau_2 = T\} \vdash \tau_1[(\textbf{Interface}\ T_1\ \tau_1)/T_1] \sqcap \tau_2 = \tau}{C \vdash (\textbf{Interface}\ T_1\ \tau_1) \sqcap \tau_2 = (\textbf{Interface}\ T\ \tau)}\ \textsc{MeetInterface}_1$$

$$\frac{C \cup \{\tau_1 \sqcap (\textbf{Interface}\ T_2\ \tau_2) = T\} \vdash \tau_1 \sqcap \tau_2[(\textbf{Interface}\ T_2\ \tau_2)/T_2] = \tau}{C \vdash \tau_1 \sqcap (\textbf{Interface}\ T_2\ \tau_2) = (\textbf{Interface}\ T\ \tau)}\ \textsc{MeetInterface}_2$$

$$\frac{C \vdash \tau_0 \sqcap \tau_0' = \tau_0'' \qquad \text{for all } i \geq 1\text{: } C \vdash \tau_i \sqcup \tau_i' = \tau_i'' \quad \tau'' = (x_1 : \tau_1'', \ldots, x_n : \tau_n'') \Rightarrow \tau_0''}{C \vdash (\,(x_1 : \tau_1, \ldots, x_n : \tau_n) \Rightarrow \tau_0) \sqcap (\,(y_1 : \tau_1', \ldots, y_n : \tau_n') \Rightarrow \tau_0') = \tau''}\ \textsc{MeetFun}$$

$$\frac{}{C \vdash \textbf{Null} \sqcap \{\overline{g : \tau'}\} = \textbf{Null}}\ \textsc{MeetNull}_1 \qquad \frac{}{C \vdash \{\overline{f : \tau}\} \sqcap \textbf{Null} = \textbf{Null}}\ \textsc{MeetNull}_2$$

$$\frac{\begin{array}{c}\forall h, \tau_1, \tau_2 : \text{if } (h : \tau_1) \in \{\overline{f : \tau_1}\} \text{ and } (h : \tau_2) \in \{\overline{g : \tau_2}\}, \text{ then } C \vdash \tau_1 \sqcap \tau_2 = \tau' \text{ for some } \tau' \\ \tau_c = \{h : \tau' \mid \exists \tau_1, \tau_2 : (h : \tau_1) \in \{\overline{f : \tau_1}\}, (h : \tau_2) \in \{\overline{g : \tau_2}\}, \text{ and } C \vdash \tau_1 \sqcap \tau_2 = \tau'\}\end{array}}{C \vdash \{\overline{f : \tau_1}\} \sqcap \{\overline{g : \tau_2}\} = \{\overline{f : \tau_1}\} \uplus \{\overline{g : \tau_2}\} \uplus \tau_c}\ \textsc{MeetObj}$$

$$\frac{(h : \tau_1) \in \{\overline{f : \tau_1}\} \quad (h : \tau_2) \in \{\overline{g : \tau_2}\} \quad C \vdash \tau_1 \sqcap \tau_2 = \bot}{C \vdash \{f : \tau_1\} \sqcap \{\overline{g : \tau_2}\} = \textbf{Null}}\ \textsc{MeetObjNull}$$

$$\frac{\tau_1 \neq \tau_2 \quad \tau_1 \in \{\textbf{bool}, \textbf{number}, \textbf{string}, \textbf{Undefined}\} \quad \tau_2 \neq (\textbf{Interface}\ T\ \tau)}{C \vdash \tau_1 \sqcap \tau_2 = \bot}\ \textsc{MeetFail}_1$$

$$\frac{\tau_1 \neq \tau_2 \quad \tau_2 \in \{\textbf{bool}, \textbf{number}, \textbf{string}, \textbf{Undefined}\} \quad \tau_1 \neq (\textbf{Interface}\ T\ \tau)}{C \vdash \tau_1 \sqcap \tau_2 = \bot}\ \textsc{MeetFail}_2$$

Figure 7: Rules for computing meets. The rules for propagating failure $\bot$ have been elided

$$\frac{\langle M, e_1\rangle \to \langle M', e_1'\rangle \quad bop \notin \{=\}}{\langle M, e_1\, bop\, e_2\rangle \to \langle M', e_1'\, bop\, e_2\rangle}\ \textsc{SearchBop}_1$$

$$\frac{\langle M, e_2\rangle \to \langle M', e_2'\rangle \quad bop \notin \{\texttt{,}, \texttt{\&\&}, \texttt{||}\}}{\langle M, v_1\, bop\, e_2\rangle \to \langle M', v_1\, bop\, e_2'\rangle}\ \textsc{SearchBop}_2$$

$$\frac{\langle M, e\rangle \to \langle M', e'\rangle}{\langle M, uop\, e\rangle \to \langle M', uop\, e'\rangle}\ \textsc{SearchUop}$$

$$\frac{\langle M, e\rangle \to \langle M', e'\rangle}{\langle M, \textbf{console.log}\,(e)\,\rangle \to \langle M', \textbf{console.log}\,(e')\,\rangle}\ \textsc{SearchPrint}$$

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

$$\frac{\langle M, e\rangle \to \langle M', e'\rangle}{\langle M, \{\ \ldots\ \texttt{,}\ f\texttt{:}e\texttt{,}\ \ldots\ \}\rangle \to \langle M', \{\ \ldots\ \texttt{,}\ f\texttt{:}e'\texttt{,}\ \ldots\ \}\rangle}\ \textsc{SearchObj} \qquad \frac{\langle M, e\rangle \to \langle M', e'\rangle}{\langle M, e\,(\overline{e})\,\rangle \to \langle M', e'\,(\overline{e})\,\rangle}\ \textsc{SearchCall}_1$$

$$\frac{\langle M, e_i\rangle \to \langle M', e_i'\rangle}{\langle M, v\,(v_1, \ldots, v_{i-1}, e_i, \ldots, e_n)\,\rangle \to \langle M', v\,(v_1, \ldots, v_{i-1}, e_i', \ldots, e_n)\,\rangle}\ \textsc{SearchCall}_2$$

$$\frac{n' = -n}{\langle M, -n\rangle \to \langle M, n'\rangle}\ \textsc{DoUMinus} \qquad \frac{b' = !b}{\langle M, !\ b\rangle \to \langle M, b'\rangle}\ \textsc{DoNot} \qquad \frac{}{\langle M, v_1\ \texttt{,}\ e_2\rangle \to \langle M, e_2\rangle}\ \textsc{DoSeq}$$

$$\frac{s = s_1 + s_2}{\langle M, s_1 + s_2\rangle \to \langle M, s\rangle}\ \textsc{DoPlusStr} \qquad \frac{n = n_1\, bop\, n_2 \quad bop \in \{+, \star, -, /\}}{\langle M, n_1\, bop\, n_2\rangle \to \langle M, n\rangle}\ \textsc{DoArith}$$

$$\frac{b = n_1\ bop\ n_2 \quad bop \in \{>, >=, <, <=\}}{\langle M, n_1\ bop\ n_2\rangle \to \langle M, b\rangle}\ \textsc{DoInequalNum}$$

$$\frac{b = s_1\ bop\ s_2 \quad bop \in \{>, >=, <, <=\}}{\langle M, s_1\ bop\ s_2\rangle \to \langle M, b\rangle}\ \textsc{DoInequalStr} \qquad \frac{b = v_1\ bop\ v_2 \quad bop \in \{===, !==\}}{\langle M, v_1\ bop\ v_2\rangle \to \langle M, b\rangle}\ \textsc{DoEqual}$$

$$\frac{}{\langle M, \textbf{true}\ \texttt{?}\ e_2\ \texttt{:}\ e_3\rangle \to \langle M, e_2\rangle}\ \textsc{DoIfThen} \qquad \frac{}{\langle M, \textbf{false}\ \texttt{?}\ e_2\ \texttt{:}\ e_3\rangle \to \langle M, e_3\rangle}\ \textsc{DoIfElse}$$

$$\frac{}{\langle M, \textbf{false}\ \texttt{\&\&}\ e_2\rangle \to \langle M, v_1\rangle}\ \textsc{DoAndFalse} \qquad \frac{}{\langle M, \textbf{true}\ \texttt{\&\&}\ e_2\rangle \to \langle M, e_2\rangle}\ \textsc{DoAndTrue}$$

$$\frac{}{\langle M, \textbf{true}\ \texttt{||}\ e_2\rangle \to \langle M, v_1\rangle}\ \textsc{DoOrTrue} \qquad \frac{}{\langle M, \textbf{false}\ \texttt{||}\ e_2\rangle \to \langle M, e_2\rangle}\ \textsc{DoOrFalse}$$

$$\frac{}{\langle M, \textbf{const}\ x = v_d\texttt{;}\ e_b\rangle \to \langle M, e_b[v_d/x]\rangle}\ \textsc{DoConst}$$

$$\frac{v \text{ printed}}{\langle M, \textbf{console.log}\,(v)\,\rangle \to \langle M, \textbf{undefined}\rangle}\ \textsc{DoPrint}$$

$$\frac{v = \textbf{function}\,(x_1\texttt{:}\tau_1, \ldots, x_n\texttt{:}\tau_n)\ t\ e}{\langle M, v\,(v_1, \ldots, v_n)\,\rangle \to \langle M, e[v_1/x_1]\ldots[v_n/x_n]\rangle}\ \textsc{DoCall}$$

$$\frac{v = \textbf{function}\ x\,(x_1\texttt{:}\tau_1, \ldots, x_n\texttt{:}\tau_n)\ \texttt{:}\tau e}{\langle M, v\,(v_1, \ldots, v_n)\,\rangle \to \langle M, e[v/x][v_1/x_1]\ldots[v_n/x_n]\rangle}\ \textsc{DoCallRec}$$

Figure 8: Small-step operational semantics of old non-imperative primitives of JavaScript.

$$\frac{\langle M, e_1\rangle \to \langle M', e_1'\rangle \quad e_1 \notin \mathit{LVal}}{\langle M, e_1 = e_2\rangle \to \langle M', e_1' = e_2\rangle}\ \textsc{SearchAssign}_1 \qquad \frac{\langle M, e_2\rangle \to \langle M', e_2'\rangle}{\langle M, lv = e_2\rangle \to \langle M', lv = e_2'\rangle}\ \textsc{SearchAssign}_2$$

$$\frac{\langle M, e_d\rangle \to \langle M', e_d'\rangle}{\langle M, \mathit{mut}\ x = e_d\,;\, e_b\rangle \to \langle M', \mathit{mut}\ x = e_d'\,;\, e_b\rangle}\ \textsc{SearchDecl} \qquad \frac{\langle M, e\rangle \to \langle M', e'\rangle}{\langle M, e\,.\,f\rangle \to \langle M', e'\,.\,f\rangle}\ \textsc{SearchGetField}$$

$$\frac{a \notin \mathit{dom}(M) \quad M' = M[a \mapsto \{\,\overline{f : v}\,\}]}{\langle M, \{\,\overline{f : v}\,\}\rangle \to \langle M', a\rangle}\ \textsc{DoObj}$$

$$\frac{a \in \mathit{dom}(M)}{\langle M, \star\, a\rangle \to \langle M, M(a)\rangle}\ \textsc{DoDeref} \qquad \frac{a \in \mathit{dom}(M) \quad M(a) = \{\,\ldots, f : v, \ldots\,\}}{\langle M, a\,.\,f\rangle \to \langle M, v\rangle}\ \textsc{DoGetField}$$

$$\frac{a \in \mathit{dom}(M)}{\langle M, \star\, a = v\rangle \to \langle M[a \mapsto v], v\rangle}\ \textsc{DoAssignVar}$$

$$\frac{a \in \mathit{dom}(M) \quad M(a) = \{\,\ldots, f : v, \ldots\,\}}{\langle M, a\,.\,f = v'\rangle \to \langle M[a \mapsto \{\,\ldots, f : v', \ldots\,\}, v'\rangle}\ \textsc{DoAssignFld}$$

$$\frac{a \notin \mathit{dom}(M) \quad M' = M[a \mapsto v_d]}{\langle M, \mathbf{var}\ x = v_d\,;\, e_b\rangle \to \langle M', e_b[\star\, a/x]\rangle}\ \textsc{DoVarDecl}$$

Figure 9: Small-step operational semantics of old imperative primitives of JakartaScript.

$$\frac{}{\langle M, <\tau> e\rangle \to \langle M, e\rangle}\ \textsc{DoCast}$$

$$\frac{}{\langle M, \mathbf{null}\,.\,f\rangle \to \langle M, \mathsf{nullerror}\rangle}\ \textsc{DoNullDeref}$$

$$\frac{}{\langle M, \mathbf{null}\,.\,f = e\rangle \to \langle M, \mathsf{nullerror}\rangle}\ \textsc{DoNullAssign}$$

Figure 10: Small-step rules for casts and null dereference errors. The error propagation rules are elided.