

## Homework 8

Due date: Tuesday, November 17, 2015

The primary purpose of this assignment is to understand the interplay between type checking and evaluation. Concretely, you will extend your JAKARTAScript interpreter from Homework 7 with a static type checker.

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 2.11.7). A program that does not compile will *not* be graded.

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

- hw08-YourNYULogin.pdf with your answers to the written questions (scanned, clearly legible handwritten write-ups are acceptable).
- hw08-YourNYULogin.scala with your answers to the coding exercises.

Replace YourNYULogin with your NYU login ID (e.g., I would submit hw08-tw47.scala, etc.). To help with managing the submissions, we ask that you rename your uploaded files in this manner.

**Getting started.** Download the code pack hw08.zip from the assignment section on the NYU classes page.

### Problem 1 Type Checking and Type Inference (16 Points)

The purpose of this exercise is to understand how the type inference rules work and what some of their limitations are. We suggest that you solve this exercise as a warm-up before you implement the type checker in Problem 2. **(7 Points)**

- (a) Use the type inference rules from Figure 3 to determine for each of the following expressions whether the expression is well-typed. If an expression is not well-typed, explain why. If it is well-typed, give the inferred type. You do not need to show the individual inference steps.

(i)

```
1 const x = 2;  
2 const y = x + 1;  
3 x * y
```

(ii)

```

1 const x = true;
2 const y = x + 1;
3 x * y

```

(iii)

```

1 const f = function f(x: number): number => number (
2     function (y: number) (x === y ? 1 : y * f(x) (y + 1))
3 );
4 f(3)

```

(iv)

```

1 const f = function f(x: number): number => number (
2     function (y: number) (x === y ? 1 : y * f(x) (y + 1))
3 );
4 f(3) === f(3)

```

(v)

```

1 const f = function f(x: number) (
2     function (y: number) (x === y ? 1 : y * f(x) (y + 1))
3 );
4 f(3) (1)

```

- (b) For each of the following programs, find concrete types for the missing parameter type annotations  $\tau_1$ ,  $\tau_2$ , and  $\tau_3$  such that the given program is well-typed according to the rules in Figure 3. If no such types exist, explain why. If you can find type annotations that make the program well-typed, what is the inferred type of `f` for your annotations? Are your chosen types the only annotations that work? If not, give at least one other choice of annotations that also makes the program well-typed. (9 Points)

(i)

```

1 const f = function (x:  $\tau_1$ )
2     (function (y:  $\tau_2$ ) (function (z:  $\tau_3$ ) (x(y(z))))))
3 const g = function (x: number) (x + 1);
4 const h = function (x: number) (x * 2);
5 f(h) (g) (3)

```

(ii)

```

1 const f = function (x:  $\tau_1$ )
2     (function (y:  $\tau_2$ ) (function (z:  $\tau_3$ ) (x(y(z))))))
3 const g = function (x:  $\tau_3$ ) (x);
4 f(g) (g)

```

$n \in Num$	numbers (double)
$s \in Str$	strings
$x \in Var$	variables
$b \in Bool ::= \mathbf{true} \mid \mathbf{false}$	Booleans
$\tau \in Typ ::= \mathbf{bool} \mid \mathbf{number} \mid \mathbf{string} \mid \mathbf{Undefined} \mid$ $(\tau_1, \dots, \tau_n) \Rightarrow \tau_0$	types
$v \in Val ::= \mathbf{undefined} \mid n \mid b \mid s \mid \mathbf{function} \ p(\overline{x:\tau}) \ t \ e \mid$ <del>typeerror</del>	values
$e \in Expr ::= x \mid v \mid uop \ e \mid e_1 \ bop \ e_2 \mid e_1 ? e_2 : e_3 \mid$ $\mathbf{const} \ x = e_1; e_2 \mid \mathbf{console.log}(e) \mid e(e_1, \dots, e_n)$	expressions
$uop \in Uop ::= - \mid !$	unary operators
$bop \in Bop ::= + \mid - \mid * \mid / \mid === \mid !== \mid < \mid > \mid <= \mid >= \mid \&\& \mid    \mid ,$	binary operators
$p ::= x \mid \epsilon$	function names
$t ::= :\tau \mid \epsilon$	return types

Figure 1: Abstract syntax

(iii)

```

1 const f = function(x:  $\tau_1$ ) (function(y:  $\tau_2$ ) (x(y)))
2 const g = function(x: bool) (x ? 1 : 0);
3 const h = function(x: number) (x + x);
4 f(g)(true) + f(h)(1)

```

## Problem 2 Statically-Typed JAKARTAScript (24 Points)

In this exercise, we will implement a strongly, statically-typed version of JAKARTAScript. We will not permit any implicit type conversions and will guarantee the absence of dynamic type errors.

We extend JAKARTAScript with a language of types  $\tau$  (see Figure 1). We also annotate function parameters with types and add optional return type annotations. Like last time, functions can take any number of parameters. We write a sequence of things using either an overbar or dots (e.g.,  $\overline{e}$  or  $e_1, \dots, e_n$  for a sequence of expressions). Note that we no longer need the special value `typeerror` to indicate a dynamic type error during evaluation. In Figure 2, we show the updated and new AST representation.

Your main task in this exercise is to implement a type checker that is very similar to a big-step interpreter. Instead of computing the value of an expression by recursively computing the value of each subexpression, we infer the type of an expression, by recursively inferring the type of each subexpression. An expression is *well-typed* if we can infer a type for it.

Given its similarity to big-step evaluation, we can formalize a type inference algorithm

```

sealed abstract class Expr extends Positional
abstract class Val extends Expr
...
type Params = List[(String, Typ)]
case class Function(p: Option[String], xs: Params,
                    t: Option[Typ], e: Expr) extends Val
Function(p, List( $\overline{(x, \tau)}$ ), t, e1) function  $p(\overline{x:\tau}) t e_1$ 

/** Types */
sealed abstract class Typ

case class TBool extends Typ
TBool bool
case class TNum extends Typ
TNum number
case class TStr extends Typ
TStr string
case class TUndefined extends Typ
TUndefined Undefined

case class TFunction(ts: List[Typ], tret: Typ) extends Typ
TFunction(List( $\overline{\tau}$ ),  $\tau_0$ )  $(\overline{\tau}) \Rightarrow \tau_0$ 

```

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.

$$\begin{array}{c}
\frac{}{\Gamma \vdash b : \mathbf{bool}} \text{TYPEBOOL} \quad \frac{}{\Gamma \vdash n : \mathbf{number}} \text{TYPERNUM} \quad \frac{}{\Gamma \vdash s : \mathbf{string}} \text{TYPESTR} \\
\\
\frac{}{\Gamma \vdash \mathbf{undefined} : \mathbf{Undefined}} \text{TYPEUNDEFINED} \quad \frac{x \in \text{dom}(\Gamma)}{\Gamma \vdash x : \Gamma(x)} \text{TYPEVAR} \\
\\
\frac{\Gamma \vdash e : \mathbf{number}}{\Gamma \vdash -e : \mathbf{number}} \text{TYPEUMINUS} \quad \frac{\Gamma \vdash e : \mathbf{bool}}{\Gamma \vdash !e : \mathbf{bool}} \text{TYPENOT} \\
\\
\frac{\Gamma \vdash e_1 : \mathbf{bool} \quad \Gamma \vdash e_2 : \mathbf{bool} \quad bop \in \{\&\&, ||\}}{\Gamma \vdash e_1 \text{ } bop \text{ } e_2 : \mathbf{bool}} \text{TYPEANDOR} \\
\\
\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1, e_2 : \tau_2} \text{TYPESEQ} \quad \frac{\Gamma \vdash e : \tau}{\Gamma \vdash \mathbf{console.log}(e) : \mathbf{Undefined}} \text{TYPEPRINT} \\
\\
\frac{\Gamma \vdash e_1 : \mathbf{string} \quad \Gamma \vdash e_2 : \mathbf{string}}{\Gamma \vdash e_1 + e_2 : \mathbf{string}} \text{TYPEPLUSSTR} \\
\\
\frac{\Gamma \vdash e_1 : \mathbf{number} \quad \Gamma \vdash e_2 : \mathbf{number} \quad bop \in \{+, *, /, -\}}{\Gamma \vdash e_1 \text{ } bop \text{ } e_2 : \mathbf{number}} \text{TYPEARITH} \\
\\
\frac{\Gamma \vdash e_d : \tau_d \quad \Gamma' = \Gamma[x \mapsto \tau_d] \quad \Gamma' \vdash e_b : \tau_b}{\Gamma \vdash \mathbf{const } x = e_d ; e_b : \tau_b} \text{TYPECONST} \\
\\
\frac{\Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \tau \quad \tau \in \{\mathbf{number}, \mathbf{string}\} \quad bop \in \{>, >=, <, <=\}}{\Gamma \vdash e_1 \text{ } bop \text{ } e_2 : \mathbf{bool}} \text{TYPEINEQUAL} \\
\\
\frac{\Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \tau \quad \tau \text{ has no function types} \quad bop \in \{===, !==\}}{\Gamma \vdash e_1 \text{ } bop \text{ } e_2 : \mathbf{bool}} \text{TYPEEQUAL} \\
\\
\frac{\Gamma \vdash e_1 : \mathbf{bool} \quad \Gamma \vdash e_2 : \tau \quad \Gamma \vdash e_3 : \tau}{\Gamma \vdash e_1 ? e_2 : e_3 : \tau} \text{TYPEIF} \\
\\
\frac{\Gamma \vdash e : (\tau_1, \dots, \tau_n) \Rightarrow \tau \quad \Gamma \vdash e_1 : \tau_1 \quad \dots \quad \Gamma \vdash e_n : \tau_n}{\Gamma \vdash e(e_1, \dots, e_n) : \tau} \text{TYPECALL} \\
\\
\frac{\Gamma' = \Gamma[x_1 \mapsto \tau_1] \dots [x_n \mapsto \tau_n] \quad \Gamma' \vdash e : \tau}{\Gamma \vdash \mathbf{function } (x_1 : \tau_1, \dots, x_n : \tau_n) e : (\tau_1, \dots, \tau_n) \Rightarrow \tau} \text{TYPEFUNCTION} \\
\\
\frac{\Gamma' = \Gamma[x_1 \mapsto \tau_1] \dots [x_n \mapsto \tau_n] \quad \Gamma' \vdash e : \tau}{\Gamma \vdash \mathbf{function } (x_1 : \tau_1, \dots, x_n : \tau_n) : \tau \text{ } e : (\tau_1, \dots, \tau_n) \Rightarrow \tau} \text{TYPEFUNCTIONANN} \\
\\
\frac{\Gamma' = \Gamma[x \mapsto \tau'] [x_1 \mapsto \tau_1] \dots [x_n \mapsto \tau_n] \quad \Gamma' \vdash e : \tau \quad \tau' = (\tau_1, \dots, \tau_n) \Rightarrow \tau}{\Gamma \vdash \mathbf{function } x (x_1 : \tau_1, \dots, x_n : \tau_n) : \tau \text{ } e : \tau'} \text{TYPEFUNCTIONREC}
\end{array}$$

Figure 3: Type checking rules for JAKARTA SCRIPT

in a similar way. In Figure 3, we define the judgment form  $\Gamma \vdash e : \tau$  which says informally, “In type environment  $\Gamma$ , expression  $e$  has type  $\tau$ .” We will implement a function

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

that corresponds directly to this judgment form. It takes as input a type environment  $\text{env}$  ( $\Gamma$ ) and an expression  $e$ , and returns a type ( $\tau$ ).

The `TYPEEQUALITY` rule is slightly informal in stating “ $\tau$  has no function types”. We intend this statement to say that the structure of  $\tau$  has no function types. The helper function `isFunctionType` is intended to return `true` iff a function type appears in the input, so this rule can be implemented by taking the negation of a call to `isFunctionType`.

To signal a type error, we will use a Scala exception

```
case class StaticTypeError(tbad: Typ, e: Expr) extends JsException
```

where `tbad` is the type that is inferred for expression  $e$ . These arguments are used to construct a useful error message. We also provide a helper function `err` to simplify throwing this exception.

It is informative to compare the typing rules with the new rules for our big-step operational semantics in Figure 4. Note that the only modification compared to the evaluation rules of Homework 7 is that we no longer use implicit type conversion functions. Moreover, we now require that the number of arguments provided in a call exactly matches the number of parameters of the called function. We rely on the type checker to detect all the cases where the evaluation may get stuck. The implementation of the interpreter will indicate such cases by throwing a `StuckError`. If you implement your type checker correctly, you will never observe a `StuckError` during the evaluation of a well-typed program.

Complete the functions `subst`, `eval`, and `typeInfer` provided in the code package. We suggest the following step-by-step order:

- (a) First, bring over the missing cases for the implementations of `subst` and `eval` from Homework 7. You will only need to make minor modifications to account for the updated representation of function expressions, which now include type annotations.
- (b) Then complete the function `typeInfer`. Start with the cases for the basic expressions. Then do the more complex cases for `Function` and `Call`.

$$\begin{array}{c}
\frac{}{v \Downarrow v} \text{EVALVAL} \quad \frac{e \Downarrow n}{-e \Downarrow -n} \text{EVALUMINUS} \quad \frac{e \Downarrow b}{!e \Downarrow !b} \text{EVALNOT} \\
\\
\frac{e_1 \Downarrow \mathbf{true} \quad e_2 \Downarrow v_2}{e_1 \&\& e_2 \Downarrow v_2} \text{EVALANDTRUE} \quad \frac{e_1 \Downarrow \mathbf{false} \quad e_2 \Downarrow v_2}{e_1 || e_2 \Downarrow v_2} \text{EVALORFALSE} \\
\\
\frac{e_1 \Downarrow \mathbf{false}}{e_1 \&\& e_2 \Downarrow \mathbf{false}} \text{EVALANDFALSE} \quad \frac{e_1 \Downarrow \mathbf{true}}{e_1 || e_2 \Downarrow \mathbf{true}} \text{EVALORTTRUE} \\
\\
\frac{e_1 \Downarrow v_1 \quad e_2 \Downarrow v_2}{e_1, e_2 \Downarrow v_2} \text{EVALSEQ} \quad \frac{e \Downarrow v \quad v \text{ printed}}{\mathbf{console.log}(e) \Downarrow \mathbf{undefined}} \text{EVALPRINT} \\
\\
\frac{e_1 \Downarrow n_1 \quad e_2 \Downarrow n_2 \quad n = n_1 + n_2}{e_1 + e_2 \Downarrow n} \text{EVALPLUSNUM} \quad \frac{e_1 \Downarrow s_1 \quad e_2 \Downarrow s_2 \quad s = s_1 + s_2}{e_1 + e_2 \Downarrow s} \text{EVALPLUSSTR} \\
\\
\frac{e_1 \Downarrow n_1 \quad e_2 \Downarrow n_2 \quad n = n_1 \text{ bop } n_2 \quad \text{bop} \in \{*, /, -\}}{e_1 \text{ bop } e_2 \Downarrow n} \text{EVALARITH} \\
\\
\frac{e_d \Downarrow v_d \quad e_b[v_d/x] \Downarrow v_b}{\mathbf{const } x = e_d; e_b \Downarrow v_b} \text{EVALCONST} \\
\\
\frac{e_1 \Downarrow n_1 \quad e_2 \Downarrow n_2 \quad b = n_1 \text{ bop } n_2 \quad \text{bop} \in \{>, >=, <, <=\}}{e_1 \text{ bop } e_2 \Downarrow b} \text{EVALINEQUALNUM} \\
\\
\frac{e_1 \Downarrow s_1 \quad e_2 \Downarrow s_2 \quad b = s_1 \text{ bop } s_2 \quad \text{bop} \in \{>, >=, <, <=\}}{e_1 \text{ bop } e_2 \Downarrow b} \text{EVALINEQUALSTR} \\
\\
\frac{e_1 \Downarrow v_1 \quad e_2 \Downarrow v_2 \quad b = (v_1 \text{ bop } v_2)}{e_1 \text{ bop } e_2 \Downarrow b} \text{EVALEQUAL} \\
\\
\frac{e_1 \Downarrow \mathbf{true} \quad e_2 \Downarrow v_2}{e_1 ? e_2 : e_3 \Downarrow v_2} \text{EVALIFTHEN} \quad \frac{e_1 \Downarrow \mathbf{false} \quad e_3 \Downarrow v_3}{e_1 ? e_2 : e_3 \Downarrow v_3} \text{EVALIFELSE} \\
\\
\frac{e_0 \Downarrow \mathbf{function}(x_1:\tau_1, \dots, x_n:\tau_n) t e}{e_1 \Downarrow v_1 \quad \dots \quad e_n \Downarrow v_n \quad e' = e[v_1/x_1] \dots [v_n/x_n] \quad e' \Downarrow v}{e_0(e_1, \dots, e_n) \Downarrow v} \text{EVALCALL} \\
\\
\frac{e_0 \Downarrow v_0 \quad v_0 = \mathbf{function } x_0(x_1:\tau_1, \dots, x_n:\tau_n) t e}{e_1 \Downarrow v_1 \quad \dots \quad e_n \Downarrow v_n \quad e' = e[v_0/x_0] \dots [v_n/x_n] \quad e' \Downarrow v}{e_0(e_1, \dots, e_n) \Downarrow v} \text{EVALCALLREC}
\end{array}$$

Figure 4: Big-step operational semantics of JAKARTASCRIPT