TEAL: Typed Easily Analysable Language

v0.4

Christoph Reichenbach

November 13, 2020

1 Introduction

Teal is a simplified language for teaching program analysis concepts. Below, we describe the syntax, static semantics, dynamic semantics, and the standard library.

2 Syntax

$$
\begin{array}{lcl}
\langle \mathit{module} \rangle & \longrightarrow & \langle \mathit{import} \rangle \star \ \langle \mathit{decl} \rangle \star \\
\\
\langle \mathit{import} \rangle & \longrightarrow & \texttt{import}\ \langle \mathit{qualified} \rangle\ \texttt{;} \\
\\
\langle \mathit{qualified} \rangle & \longrightarrow & \mathit{id} \\
 & \mid & \langle \mathit{qualified} \rangle\ \texttt{::}\ \mathit{id} \\
\\
\langle \mathit{decl} \rangle & \longrightarrow & \langle \mathit{vardecl} \rangle \\
 & \mid & \texttt{fun}\ \mathit{id}\ \texttt{(}\ \langle \mathit{formals} \rangle?\ \texttt{)}\ \langle \mathit{opttype} \rangle\ \texttt{=}\ \langle \mathit{stmt} \rangle \\
\\
\langle \mathit{vardecl} \rangle & \longrightarrow & \texttt{var}\ \mathit{id}\ \langle \mathit{opttype} \rangle \\
 & \mid & \texttt{var}\ \mathit{id}\ \langle \mathit{opttype} \rangle\ \texttt{:=}\ \langle \mathit{expr} \rangle \texttt{;} \\
\\
\langle \mathit{formals} \rangle & \longrightarrow & \mathit{id}\ \langle \mathit{opttype} \rangle \\
 & \mid & \mathit{id}\ \langle \mathit{opttype} \rangle\ \texttt{,}\ \langle \mathit{formal} \rangle \\
\langle \mathit{formal} \rangle & \langle \mathit{vardecl} \rangle & \\
\\
\langle \mathit{opttype} \rangle & \longrightarrow & \texttt{:}\ \langle \mathit{type} \rangle \\
 & \mid & \varepsilon \\
\\
\langle \mathit{type} \rangle & \longrightarrow & \texttt{int} \mid \texttt{string} \mid \texttt{any} \\
 & \mid & \texttt{array [}\ \langle \mathit{type} \rangle\ \texttt{]} \\
\\
\langle \mathit{block} \rangle & \longrightarrow & \texttt{\{}\ \langle \mathit{stmt} \rangle \star\ \texttt{\}} \\
\\
\langle \mathit{expr} \rangle & \longrightarrow & \langle \mathit{expr} \rangle\ \langle \mathit{binop} \rangle\ \langle \mathit{expr} \rangle \\
 & \mid & \texttt{not}\ \langle \mathit{expr} \rangle \\
 & \mid & \texttt{(}\ \langle \mathit{expr} \rangle\ \langle \mathit{opttype} \rangle\ \texttt{)} \\
 & \mid & \langle \mathit{expr} \rangle\ \texttt{[}\ \langle \mathit{expr} \rangle\ \texttt{]} \\
 & \mid & \mathit{id}\ \texttt{(}\ \langle \mathit{actuals} \rangle?\ \texttt{)} \\
 & \mid & \texttt{[}\ \langle \mathit{actuals} \rangle?\ \texttt{]} \\
 & \mid & \texttt{new}\ \langle \mathit{type} \rangle\ \texttt{(}\ \langle \mathit{expr} \rangle\ \texttt{)} \\
 & \mid & \mathit{int} \mid \mathit{string} \mid \texttt{null} \\
 & \mid & \mathit{id} \\
\\
\langle \mathit{actuals} \rangle & \longrightarrow & \mathit{expr} \\
 & \mid & \mathit{expr},\ \langle \mathit{actuals} \rangle \\
\\
\langle \mathit{binop} \rangle & \longrightarrow & \texttt{+} \mid \texttt{-} \mid \texttt{*} \mid \texttt{/} \mid \texttt{\%} \\
 & \mid & \texttt{==} \mid \texttt{!=} \mid < \mid <= \mid >= \mid > \\
 & \mid & \texttt{or} \mid \texttt{and} \\
\\
\langle \mathit{stmt} \rangle & \longrightarrow & \langle \mathit{vardecl} \rangle \\
 & \mid & \langle \mathit{expr} \rangle\ \texttt{;} \\
 & \mid & \langle \mathit{expr} \rangle\ \texttt{:=}\ \langle \mathit{expr} \rangle\ \texttt{;} \\
 & \mid & \langle \mathit{block} \rangle \\
 & \mid & \texttt{if}\ \langle \mathit{expr} \rangle\ \langle \mathit{block} \rangle\ \texttt{else}\ \langle \mathit{block} \rangle \\
 & \mid & \texttt{if}\ \langle \mathit{expr} \rangle\ \langle \mathit{block} \rangle \\
 & \mid & \texttt{while}\ \langle \mathit{expr} \rangle\ \langle \mathit{block} \rangle \\
 & \mid & \texttt{return}\ \langle \mathit{expr} \rangle\ \texttt{;}
\end{array}
$$

3 Semantics and Failure

The definitions of the static and dynamic semantics use a notion of *failure*. Failure in Teal means that compilation (statically) or execution (dynamically) *may* abort, and otherwise, program semantics are undefined.

4 Static Semantics

Teal-0 enforces the following static constraints, and each violation of such a constraint is a *static failure*:

1. **Name analysis**: Teal-0 performs name analysis and enforces the following name-related constraints:
   (a) Each variable use binds to exactly one variable definition
   (b) Each function use binds to exactly one function definition
   (c) No variable is used within its defining module *prior* to its declaration, where *prior* refers to the pre-order traversal of the AST.
2. **No initialisation of module-level variables**: Syntactically, all variable declarations **var** `x;` may contain an optional initialisation expression:

   var `x := ⟨expr⟩ ;`

   We statically disallow variable initialisations if this production is derived via $\langle \textit{module} \rangle \rightarrow \langle \textit{decl} \rangle \rightarrow \langle \textit{vardecl} \rangle$.
3. All *int* literals ℓ are representable in the set of signed two's complement 64 bit integers $\mathbb{I}_{64}$
4. All *string* literals are representable as Unicode character strings and do not contain supplementary [1] characters
5. All assignments $e_1 := e_2$ subject e_1 to the *lvalue restriction*, in that e_1 must be one of
   (a) *id*
   (b) e`[`e'`]`
6. (Teal-0 enforces no type checking constraints.)

4.1 Name Analysis

1. All names share the same scope:
   - Variables
   - Functions
2. The following are *variable definitions*:
   (a) The following code fragments declare the underlined identifiers as variables that are visible to all sibling AST nodes and their descendants:
      i. **import** `. . . ::x;`
      ii. **var** `x . . . ;`
   (b) The following code fragments declare the underlined identifiers as variables that are visible to all descendant AST nodes:
      i. **fun** `f (x1, . . . , xn) . . .`
3. The following are *function definitions*:

[1] Characters encoded with more than 16 bits.

(a) The following code fragments declare the underlined identifiers as functions that are visible to (a) the declaring AST node, (b) its sibling AST nodes, and (c) the descendants of (a) and (b):

i. `fun f (x1, ..., xn) ...`

4. Collectively, these *definitions* are *name definitions*:
   - Variable definitions
   - Function definitions

5. All other occurrences of identifiers are *name uses* and bind to all declared identifiers with the same name:

   (a) `f (...)` is a function use

   (b) All other name uses are *variable uses*

6. For purposes of the dynamic semantics, each name definition introduces a globally unique name that we refer to as:
   - *Variable*, for variable definitions
   - *Function*, for function definitions

5 Dynamic Semantics

We define dynamic semantics largely informally. In particular, we use a notion of *current state* S, which tracks input, output, and variable state:

1. Each variable *may* be bound to a *storage*
2. Each storage *may* be bound to a single *object*
3. $\textit{store}_S(x, o)$, where x is a variable and o an object in a current state S, yields a *new* state S' to which S is the *previous state* and $\langle x,o \rangle$ is the *state update*
4. In state S, $\textit{load}_S(x)$, where x is a variable, yields the same result as $\textit{load}_{S'}(x)$ in the previous state S', unless the state update to S was $\langle x, o \rangle$, in which case $\textit{load}_S(x)$ yields o
5. $\textit{load}_S(x)$ on a state S that has no previous state yields **null**

Note that our notion of *current state* does not track fresh objects.

5.1 Teal Objects

1. Teal evaluation computes *objects* (corresponding to *values* in operational semantics)
2. Teal objects all have *reference semantics*
3. Teal has the following built-in *values*, which are *immutable objects*:

   (a) The distinguished value **null**

   (b) All 64 bit signed two's complement integer values (*integers*), such as **0**, **1**, ...

   (c) All Unicode character strings that do not contain supplementary characters (*strings*), such as "foo", "bar", ...

4. Teal-0 has the following additional built-in objects:

   (a) *Arrays*, which are mutable objects with the following properties:

   i. A fixed (per object) nonnegative integer *length*, *len*(arr)

   ii. An index range *irange*(arr) = $\{i \in \mathbb{N} | 0 \leq i < \textit{len}(\text{arr})\}$

   iii. Mutable *elements* arr_i where $i \in$ *irange*(arr) that behave as *variables*

5.2 Teal Dynamic Types

1. We define semantics with the help of *dynamic types* that categorise Teal objects
2. We use notation $o : \tau$ to indicate that object o has type τ
3. We define the following dynamic types for objects o in some state S:
   (a) $o : \textsc{Any}$
   (b) $o : \textsc{Int} \iff o$ is an integer
   (c) $o : \textsc{String} \iff o$ is a string
   (d) $o : \textsc{Array}[\tau] \iff o$ is an array, and for all $i \in$ *irange*(o), $\mathit{load}_S(o_{[i]}) : \tau$
   (e) $o : \tau_\perp \iff o : \tau$ or $o = \perp$

5.3 Teal Dynamic Equality

1. Two teal objects v_1 and v_2 are *Teal-equal* if any of the following hold:
   (a) v_1 and v_2 are both **null**
   (b) $v_1 : \textsc{Int}$ and $v_2 : \textsc{Int}$ and $v_1 = v_2$
   (c) $v_1 : \textsc{String}$ and $v_2 : \textsc{String}$ and $v_1 = v_2$
   (d) v_1 and v_2 have the same *identity*, according to their references

5.4 Expression Evaluation

1. We write $\langle S, e\rangle \downarrow \langle S', o\rangle$ if expression e in state S evaluates to object o and produces state S'
2. $e \downarrow o$ is short for $\langle S, e\rangle \downarrow \langle S', o\rangle$ and $S = S'$
3. For evaluating e in state S we consider the following cases:
   (a) $e_1 \odot e_2$: $\langle S, e\rangle \downarrow \langle S_2, v\rangle$ where
      - $\langle S, e_1\rangle \downarrow \langle S_1, v_1\rangle$
      - $\langle S_1, e_2\rangle \downarrow \langle S_2, v_2\rangle$
      - Consider the following cases for $\odot$:
        i. $\odot =$ `==`: $v = \mathbf{1}$ iff v_1 is *Teal-equal* to v_2, otherwise $\mathbf{0}$
        ii. $\odot =$ `!=`: $v = \mathbf{0}$ iff v_1 is *Teal-equal* to v_2, otherwise $\mathbf{1}$
        iii. Otherwise, if not $v_1 : \textsc{Int}$ or not $v_2 : \textsc{Int}$, evaluation fails
        iv. Otherwise consult the table below and fail if the **Requirements** are not met:

$\odot$	**Requirement**	$v =$
`+`	$v \in \mathbb{I}_{64}$	$v_1 + v_2$
`-`	$v \in \mathbb{I}_{64}$	$v_1 - v_2$
`*`	$v \in \mathbb{I}_{64}$	$v_1 \times v_2$
`/`	$v_2 \neq 0$ and $v \in \mathbb{I}_{64}$	$\frac{v_1}{v_2}$
`%`	$v_2 > 0$ and $v \in \mathbb{I}_{64}$	$v_1 \bmod v_2$
$<$		$\mathbf{1} \iff v_1 < v_2,$ otherwise $\mathbf{0}$
$<=$		$\mathbf{1} \iff v_1 \leq v_2,$ otherwise $\mathbf{0}$
$>=$		$\mathbf{1} \iff v_1 \geq v_2,$ otherwise $\mathbf{0}$
$>$		$\mathbf{1} \iff v_1 > v_2,$ otherwise $\mathbf{0}$
and		$\mathbf{0} \iff v_1 = \mathbf{0}$ or $\mathbf{0} = v_2,$ otherwise $\mathbf{1}$
or		$\mathbf{0} \iff v_1 = \mathbf{0} = v_2,$ otherwise $\mathbf{1}$

   (b) `not` e': Let $\langle S, e'\rangle \downarrow \langle S', v\rangle$, then

- $\langle S, e\rangle \downarrow \langle S', \mathbf{1}\rangle$ iff v = $\mathbf{0}$
- $\langle S, e\rangle \downarrow \langle S', \mathbf{0}\rangle$ otherwise

(c) $(e' : \tau)$: same as e'

(d) (e'): same as e'

(e) e_1 `[`e_2`]`: Let

- $\langle S, e_1\rangle \downarrow S_1, a$
- $\langle S_1, e_2\rangle \downarrow S_2, v$, then:
- If not a : $\textsc{Array}[\tau_\perp]$: failure
- If not v : $\textsc{Int}$: failure
- If not $v \in$ *irange*(a): failure
- Otherwise, $\langle S, e\rangle \downarrow S_2, \mathit{load}_{S_2}(a_{[v]})$

(f) f `(`e_1`, ..., `e_k`)`: Informally:

i. Evaluation first evaluates arguments e_1 through e_k, including side effects
ii. Evaluation backs up all current local variable states in the current activation record
iii. Evaluation binds the results of the previous execution to the formal parameter variables of f, in the same order
iv. Evaluation executes the body of f until it reaches a **return** e_r statement or the end of the function.
    - If Evaluation reaches the end of the function, then this is equivalent to reaching **return** `null`.
v. At this point evaluation evaluates e_r, remembers it as v_r restores the local variables stored in step 2 above, and yields v_r.

(g) `[`e_1`, ..., `e_k`]`: $\langle S,e\rangle \downarrow \langle S'_k,o\rangle$, where:

i. $S'_0 = S$
ii. o : $\textsc{Array}[\textsc{Any}]$ is a fresh array
iii. *len*(o) $= k$
iv. for all $i \in$ *irange*(o):
    - $\langle S_{i-1},e_i\rangle \downarrow \langle S_i, v_i\rangle$
    - $S'_i = \mathit{store}_{S_i}(o_{[i]}, v_i)$

(h) **new** τ `(`e'`)`:

i. If $\tau = \textsc{Array}[\tau']$ for some τ' and $\langle S,e'\rangle \downarrow \langle S',v\rangle$ where ℓ : $\textsc{Int}$ and $\ell \geq 0$, then $\langle S,e\rangle \downarrow \langle S',o\rangle$, where:
    - o : $\textsc{Array}[\textsc{Any}]$ is a fresh array
    - *len*(o) $= \ell$
ii. Otherwise evaluation fails

(i) *int*: $e \downarrow v$ where v is the corresponding integer in $\textsc{Int}$

(j) *string*: $e \downarrow v$ where v is the corresponding string in $\textsc{String}$ (not including leading/trailing double quotes)

(k) `null`: $e \downarrow$ **null**

(l) *id*: let x the variable assigned to e, then $e \downarrow \mathit{load}_S(x)$

5.5 Statement Execution

1. We write $s{:}S{\Downarrow}S'$ if statement s transforms state S into state S'
2. For a given state S, we distinguish between statements s (ignoring optional type annotations):
   (a) **var** x `:=` e`;`: Let $\langle S, e\rangle \downarrow \langle S', o\rangle$, then $s{:}S{\Downarrow}\mathit{store}_{S'}(x, o)$
   (b) e`;`: Let $\langle S, e\rangle \downarrow \langle S', o\rangle$, then $s{:}S{\Downarrow}S'$
   (c) e_1 `:=` e_2`;`: Let
      i. $\langle S, e_2\rangle \downarrow \langle S', o\rangle$
      ii. Distinguish e_1 as follows:
         A. *id*: then $s{:}S{\Downarrow}\mathit{store}_{S'}(e_1, o)$
         B. e_3`[`e_4`]`: Let
            - $\langle S', e_3\rangle \downarrow S_1, a$
            - $\langle S_1, e_4\rangle \downarrow S_2, v$, then:
            - If not $a : \textsc{Array}[\tau_\perp]$: failure
            - If not $v : \textsc{Int}$: failure
            - If not $v \in \mathit{irange}(a)$: failure
            - Otherwise, $s{:}S{\Downarrow}\mathit{store}_{S_2}(a_v, o)$
   (d) `{` s_1 `. . .` s_k `}`: $s{:}S{\Downarrow}S_k$, where
      - $S_0 = S$
      - for all $i \in \{1, \ldots, k\}$: $s_i{:}S_{i-1}{\Downarrow}S_i$
   (e) **if** e s_1 **else** s_2: Let $\langle S, e\rangle \downarrow \langle S', o\rangle$.
      i. If $o = \mathbf{0}$ then $s{:}S{\Downarrow}S_2$, where $s_2{:}S'{\Downarrow}S_2$
      ii. Otherwise $s{:}S{\Downarrow}S_2$, where $s_1{:}S'{\Downarrow}S_1$
   (f) **if** e s_1: same as **if** e s_1 **else** `{}`
   (g) **while** e s': Let $\langle S, e\rangle \downarrow \langle S', o\rangle$.
      i. If $o = \mathbf{0}$ then $s{:}S{\Downarrow}S'$.
      ii. Otherwise $s{:}S{\Downarrow}S_w$, where $\{s'; s\}{:}S'{\Downarrow}S_w$
   (h) **return** e`;`: See the evaluation of function calls in Section **??**.

6 Teal Standard Library

Teal predefines a number of functions:

Builtin function name	Type or Type Scheme	Semantics
`__builtin_int_add`	$\text{INT} \times \text{INT} \to \text{INT}$	Section 5.4: `+`
`__builtin_int_sub`	$\text{INT} \times \text{INT} \to \text{INT}$	Section 5.4: `-`
`__builtin_int_mul`	$\text{INT} \times \text{INT} \to \text{INT}$	Section 5.4: `*`
`__builtin_int_div`	$\text{INT} \times \text{INT} \to \text{INT}$	Section 5.4: `/`
`__builtin_int_mod`	$\text{INT} \times \text{INT} \to \text{INT}$	Section 5.4: `%`
`__builtin_int_eq`	$\forall\alpha.\alpha \times \alpha \to \text{INT}$	Section 5.4: `==`
`__builtin_int_neq`	$\forall\alpha.\alpha \times \alpha \to \text{INT}$	Section 5.4: `!=`
`__builtin_int_leq`	$\text{INT} \times \text{INT} \to \text{INT}$	Section 5.4: `<=`
`__builtin_int_geq`	$\text{INT} \times \text{INT} \to \text{INT}$	Section 5.4: `>=`
`__builtin_int_lt`	$\text{INT} \times \text{INT} \to \text{INT}$	Section 5.4: `<`
`__builtin_int_gt`	$\text{INT} \times \text{INT} \to \text{INT}$	Section 5.4: `>`
`__builtin_int_logical_and`	$\text{INT} \times \text{INT} \to \text{INT}$	Section 5.4: `and`
`__builtin_int_logical_or`	$\text{INT} \times \text{INT} \to \text{INT}$	Section 5.4: `or`
`concat`	$\text{STRING} \times \text{STRING} \to \text{STRING}$	Concatenates two strings
`print`	$\forall\alpha.\alpha \to \text{ANY}$	Prints out a string representation of its parameter
`read`	$() \to \text{STRING}$	Reads a text line from the user
`string_to_int`	$\text{STRING} \to \text{INT}$	STRING to INT conversion, or failure if not possible
`int_to_string`	$\text{INT} \to \text{STRING}$	Map INT to string representation
`can_convert_to_int`	$\text{STRING} \to \text{INT}$	Yields **0** iff `string_to_int` would fail, **1** otherwise
`array_length`	$\forall\alpha.\text{ARRAY}[\alpha] \to \text{INT}$	maps arr to *len*(arr)