Specification of Dynamic TypeScript §1—2021 edition

Dorcas Tan, Wei Feng

National University of Singapore
School of Computing

April 23, 2021

1 Introduction

Dynamic TypeScript §1 is a variant of TypeScript that makes use of dynamic type checking instead of the usual static type checking. It parallels Source §1 in scope, maintaining most of its syntax, but it also introduces TypeScript's type annotations to allow for type checking.
The main changes are summarised below.

1.1 Type Annotations

Constant declarations can have optional type annotations. For functions, all parameter types and return types must be annotated. (Note that this means that single parameters in lambda expressions must also be enclosed in brackets.)

1.2 Polymorphic Functions

Functions can also be declared with type parameters, allowing them to be applied to arguments of different types as long as they meet certain constraints. When applying a generic function, the type arguments (i.e. instantiation of type parameters) must be specified.
Here is an example of a polymorphic function:

```
function repeat<T>(val: T, n: number, f: (x: T) => T): T {
   return n === 0
       ? val
       : repeat<T>(f(val), n - 1, f);
}
repeat<number>(0, 10, (x: number): number => x + 1);
repeat<string>("", 3, (s: string): string => s + "abc");
```

1.3 Differences from Source §1

Apart from the obvious semantic differences due to type checking, Dynamic TypeScript §1 does not support import and export directives.

2 Syntax

The syntax rules for Dynamic TypeScript §1 are as follows:

program	::=	*statement* ...	program
statement	::=	**const** *name* **=** *expr* (**;**)	constant declaration
	|	**const** *name*: *type* **=** *expr* (**;**)	annotated constant declaration
	|	**function** *name* **(** *parameters* **)** : *type block*	function declaration
	|	**function** *name* **<***type-params***>(** *parameters* **)** : *type block*	generic function declaration
	|	**return** *expr* (**;**)	return statement
	|	*if-statement*	conditional statement
	|	*block*	block statement
	|	*expr* (**;**)	expression statement
type	::=	**number**	primitive number type
	|	**boolean**	primitive boolean type
	|	**string**	primitive string type
	|	*name*	type reference
	|	**(** *parameters* **) =>** *type*	function type
	|	**<***type-params***>(** *parameters* **) =>** *type*	generic function type
parameters	::=	ϵ | *name*:*type* (**,** *name*:*type*) ...	function parameters
type-params	::=	*name* (**,** *name*) ...	type parameters
if-statement	::=	**if (** *expr* **)** *block* **else** (*block* | *if-statement*)	conditional statement
block	::=	**{** *statement* ... **}**	block statement
expr	::=	*number*	primitive number expression
	|	**true** | **false**	primitive boolean expression
	|	*string*	primitive string expression
	|	*name*	name expression
	|	*expr binary-operator expr*	binary operator combination
	|	*unary-operator expr*	unary operator combination
	|	*expr* **(** *expressions* **)**	function application
	|	*expr* **<***types***>(** *expressions* **)**	generic function application
	|	**(** *parameters* **) :** *type* **=>** *expr*	lambda expression (expr. body)
	|	**<***type-params***>(** *parameters* **) :** *type* **=>** *expr*	
	|	**(** *parameters* **) :** *type* **=>** *block*	lambda expression (block body)
	|	**<***type-params***>(** *parameters* **) :** *type* **=>** *block*	
	|	*expr* **?** *expr* **:** *expr*	conditional expression
	|	**(** *expr* **)**	parenthesised expression
binary-operator	::=	**+** | **-** | ***** | **/** | **%** | **===** | **!==** | **>** | **<** | **>=** | **<=** | **&&** | **||**	binary operator
unary-operator	::=	**!** | **-**	unary operator
expressions	::=	ϵ | *expr* (**,** *expr*) ...	argument expressions
types	::=	*type* (**,** *type*) ...	type arguments

Restrictions

Dynamic TypeScript §1 contains most of the same restrictions as Source §1.
In particular:

- There cannot be any newline character between (*name* | **(** *parameters* **)**) and **=>** in function definition expressions.
- Return statements are not allowed to be empty, i.e. the statement `return;` is invalid.
- Implementations are allowed to treat function declaration as syntactic sugar for constant declaration. Dynamic TypeScript §1 programmers need to make sure that functions are not called before their corresponding function declaration is evaluated.

The key differences are:

- Semicolons are not required.
- If there is a newline character between **`return`** and *expression* in return statements, then **`return`** is treated as an empty return statement (which is not allowed).

3 Dynamic Type Checking

3.1 Infrastructure

Runtime Type Information

Expressions evaluate to number, boolean, string, or function values. Every value is also tagged with its type to facilitate type checking. For example, the expression **`1`** is evaluated to $(1, \texttt{number})$. Note that these $(\textit{value}, \textit{type})$ pairs will be referred to as *typed values*.
Implementations generate error messages when unexpected types are used, as defined in section 3.2.

Type Environment

In order to deal with type references, we introduce a type environment, which maps names to types. The types in the type environment must not be type references, but they can *contain* type references—for example, the parameters and return type of a generic function type like **`<T>(x:T) => T`** can be references to its type parameters.

3.2 Evaluation

Evaluation $\mapsto_{TS_D}$ is a quinary (5-ary) relation that maps an expression to a value and type in the context of an environment and type environment. Formally,

$$(\text{Env}, \text{TypeEnv}) \vdash \text{Dynamic TypeScript §1} \mapsto_{TS_D} (\text{Value, Type})$$

where Env and TypeEnv are defined as follows:

$$\begin{aligned} \text{Env} &: \text{Name} \mapsto (\text{Value, Type}) \\ \text{TypeEnv} &: \text{Name} \mapsto \text{Type} \end{aligned}$$

Types

Since type names are never re-assigned, each type expression in a Dynamic TypeScript §1 program is mapped directly to a runtime type.
Primitive types are mapped to their runtime equivalents (i.e. runtime number, runtime boolean, runtime string). A runtime undefined type is used as the result type for non-value-producing statements (e.g. constant declarations).
A runtime function type consists of a list of type parameters, a list of parameter types (which are themselves runtime types), and a return type (which is a runtime type). Dynamic TypeScript §1 function types are mapped to their runtime equivalents by recursively converting the parameter and return types.
Implementations should also allow for runtime type references to deal with generic function types. However, type names are resolved to their actual types where possible (the exact behaviour is described in the last paragraph of section 3.2).

Primitive expressions

As mentioned above, numbers, booleans, and strings are evaluated to their respective semantic values, with the types being **`number`**, **`boolean`**, or **`string`** respectively.

Primitive operators

When evaluating unary and binary operator combinations, operand types are first checked according to the tables below. A error message specifying the expected type is produced if the types are not equal.

operator	operand 1	operand 2	result
`+`	number	number	number
`+`	string	string	string
`-`	number	number	number
`*`	number	number	number
`/`	number	number	number
`%`	number	number	number
`===`	number	number	bool
`===`	string	string	bool
`!==`	number	number	bool
`!==`	string	string	bool
`>`	number	number	bool
`>`	string	string	bool
`<`	number	number	bool
`<`	string	string	bool
`>=`	number	number	bool
`>=`	string	string	bool
`<=`	number	number	bool
`<=`	string	string	bool
`&&`	bool	any	any
`||`	bool	any	any
`!`	bool		bool
`-`	number		number

Conditional expressions

Conditional expressions are of the form *predicate*`?`*consequent*`:`*alternative*. The result of evaluating *predicate* (in the context of the current Env and TypeEnv) must have type **`boolean`**. However, the results of evaluating *consequent* and *alternative* (in the current Env and TypeEnv) can have different types.

Constant declarations

Constant declarations add a name-value binding to Env as follows:

$$\frac{E \mapsto_{TS_D} (v_1, t_1) \qquad (\text{Env}[x \leftarrow (v_1, t_1)], \text{TypeEnv}) \vdash S; \mapsto_{TS_D} (v_2, t_2)}{(\text{Env, TypeEnv}) \vdash \texttt{const}\ x = E; S;\ \mapsto_{TS_D} (v_2, t_2)}$$

In the case of annotated constant declarations, the type annotation is first converted to a runtime type $t_{expected}$. $t_{expected}$ is then checked to ensure it is **valid** in the current TypeEnv, which means that if $t_{expected}$ contains a reference to a name y, there must be some type $t_{resolved}$ such that TypeEnv(y) = $t_{resolved}$.
Finally, implementations must check that $t_{expected}$ and t_1 are **equal**. At a basic level, two types are equal if and only if they are both primitive types (number/boolean/string) and have the same type, or they are function types and all of their parameter and return types are equal.
A formal definition of validity and type equality is provided in section 3.2.

Function declarations

Like in Source §1, function declarations are treated as constant declarations with a lambda expression value. A lambda expression is evaluated to a closure, i.e. (*parameters*, *body*, *environment*, *type-environment*), and a function type.
Implementations should first ensure that all parameter and return types are annotated, and show an error message otherwise.
Implementations should also check that the runtime function type corresponding to the given type annotations is *valid* in TypeEnv (as defined above).

Function applications

Only values with function types can be applied, or in other words:

$$\frac{(\textit{Env}, \textit{TypeEnv}) \vdash E \mapsto_{TS_D} (v, t)}{(\textit{Env}, \textit{TypeEnv}) \vdash E(E_1, \ldots, E_n) \mapsto_{TS_D} \textit{Error}} \text{ if } t \text{ is not a function type}$$

If t is a function type, then all arguments are evaluated to typed values before type checking takes place.
First, the number of arguments n must be *equal* to the number of parameters in the function type t. Given that $t = (t_{e1}, \ldots, t_{em}) \rightarrow t_r$, $n = m$.
Second, the type of each argument must be *equal* to the corresponding parameter type. Given that $E_1, \ldots, E_n$ evaluate to $(v_1, t_1), \ldots, (v_n, t_n)$, we have t_{ei} is equal to t_i for $i \in [1, n]$.
If none of the above checks results in a type error, the body of the function is evaluated in an extension of the closure's environment, where the names of function parameters are bound to their respective arguments (typed values). The result of the function application is the resulting value from the return statement.
The type of the function's return value must be *equal* to the function's return type. Otherwise, an error message is shown.

Tail call optimisation
Dynamic TypeScript §1 supports proper tail calls. If a function's return statement only contains a function application, the function's environment is reused in the next .
In this case, type checking proceeds in the same way as normal function applications.

Polymorphic (Generic) Function Declaration

This section concerns lambda expressions and function declarations with type parameters. For example:

$$\texttt{<}P_1, \ldots, P_k\texttt{>(}x_1 : T_1, \ldots, x_n : T_n\texttt{):}\quad T_r\ \texttt{=>}\ \texttt{\{}\ S\ \texttt{\}}$$

Note that $T_1, \ldots, T_n, T_r$ can also be generic function types whose parameter and return types contain references to $P_1, \ldots, P_k$.
Hence, we modify the definition of **validity** as follows:

- A type's validity is checked in the context of a type environment and a temporary set of types $Types$, which contains the names of type parameters from the function type.
- When checking the validity of a generic function type, first add the type parameters $P_1, \ldots, P_k$ to a copy of $Types$ to get $Types_{new}$.
- Then recursively check that each of $T_1, \ldots, T_n, T_r$ is valid given the new set $Types_{new}$ and the original *TypeEnv*. A type is valid if, whenever it contains a reference to a name y, either y is found in $Types_{new}$, or y is found in the type environment (there exists $t_{resolved}$ such that *TypeEnv*(y) = $t_{resolved}$).

We also need to modify the definition of **type equality**. Note that the runtime function types from the following two functions should be equal:

```
<T>(x: T): <R>(y: T) => R => x
<S>(x: S): <P>(y: S) => P => x
```

Hence, we add a temporary "type environment" $TTEnv$ that maps type names to their position in the type hierarchy. A position is a pair (a, b) where a is the index of the type in the list of type parameters and b is the number of nested levels of hierarchy. [1]
Hence, two types t_1 and t_2 are equal in the context of $TTEnv_1$ and $TTEnv_2$ if and only if:

- They are the same primitive type (number/boolean/string).
- They are both type references and the type names that they reference have the same position in their respective $TTEnv$.
- They are both function types and their parameter and return types are equal in the context of $TTEnv_1$ and $TTEnv_2$.
- They are both generic function types with the same number of type parameters, and their parameter and return types are equal in the context of $TTEnv_1^{new}$ and $TTEnv_2^{new}$. The new temporary type environments $TTEnv_i^{new}$ are constructed by extending $TTEmv_i$ with a mapping of each type parameter to its position. Note that this allows previous mappings of the same type name to be shadowed. Also note that the function must keep track of the current level of hierarchy. This level is incremented when the new temporary type environments are used, i.e. when checking a generic functions' parameter and return types after constructing a temporary type environment. The level can also (optionally) be incremented when checking a non-generic function's parameter and return types.

Finally, before constructing any type (e.g. the expected type for constant declarations), type references must first be **resolved** to their actual values in the current *TypeEnv*. In other words: for every type reference y in the type to be resolved, y either refers to a type parameter or a type in *TypeEnv*. If it refers to a type parameter, the reference y is left as it is. However, if it references a type in *TypeEnv*, the y is "replaced" by *TypeEnv(y)*. (Runtime types are immutable, so "replacement" involves constructing a new resolved runtime type.)

[1] In the above example, the type annotation T for the parameter x references a type at position (0, 0) - since T is the first type parameter in `<T>` and it is at the top level. On the other hand, the type annotation R for the return type of the return type references a type at position (0, 1), since R is the first type parameter in `<R>` and the generic function where it is defined is nested one level deep.

Polymorphic (Generic) Function Application

Consider the following application of a function `foo` with type `<T>(y: T, f: (x: T) => T) => T`:

```
foo<number>(1, (x: number): number => x + 1)
```

When applying a generic function, we check that the number of arguments and type arguments are correct.
We then create a(nother) temporary type environment where each type parameter is bound to its type argument, and resolve the function parameter types (i.e. runtime types of `T` and `(x: T) => T`) to their actual types $t_{expected,1}, \ldots, t_{expected,n}$ (i.e. the runtime types of `number` and `(x: number) => number`).
We then check that each expected parameter type is *equal* to its corresponding argument type.
Next, the environment and type environment *of the closure* are extended as follows:

- *TypeEnv*: Add a binding of each type parameter name to the type argument.
- *Env* (no change): Add a binding of each parameter name to the argument (v, t).

The function body is evaluated in the context of the new Env and TypeEnv.
Finally, the expected return type is resolved in the context of the new type environment (which contains the bindings for the type parameters). The type of the function's return value must be equal to the expected return type.