# Understanding TypeScript

Gavin Bierman[1,⋆], Martín Abadi[2], and Mads Torgersen[2]

[1] Oracle Gavin.Bierman@oracle.com
[2] Microsoft {abadi,madst}@microsoft.com

**Abstract.** TypeScript is an extension of JavaScript intended to enable easier development of large-scale JavaScript applications. While every JavaScript program is a TypeScript program, TypeScript offers a module system, classes, interfaces, and a rich gradual type system. The intention is that TypeScript provides a smooth transition for JavaScript programmers—well-established JavaScript programming idioms are supported without any major rewriting or annotations. One interesting consequence is that the TypeScript type system is not statically sound by design. The goal of this paper is to capture the essence of TypeScript by giving a precise definition of this type system on a core set of constructs of the language. Our main contribution, beyond the familiar advantages of a robust, mathematical formalization, is a refactoring into a safe inner fragment and an additional layer of unsafe rules.

## 1 Introduction

Despite its success, JavaScript remains a poor language for developing and maintaining large applications. TypeScript is an extension of JavaScript intended to address this deficiency. Syntactically, TypeScript is a superset of EcmaScript 5, so every JavaScript program is a TypeScript program. TypeScript enriches JavaScript with a module system, classes, interfaces, and a static type system. As TypeScript aims to provide lightweight assistance to programmers, the module system and the type system are flexible and easy to use. In particular, they support many common JavaScript programming practices. They also enable tooling and IDE experiences previously associated with languages such as C♯ and Java. For instance, the types help catch mistakes statically, and enable other support for program development (for example, suggesting what methods might be called on an object). The support for classes is aligned with proposals currently being standardized for EcmaScript 6.

The TypeScript compiler checks TypeScript programs and emits JavaScript, so the programs can immediately run in a huge range of execution environments. The compiler is used extensively in Microsoft to author significant JavaScript applications. For example, recently[3] Microsoft gave details of two substantial TypeScript projects: Monaco, an online code editor, which is around 225kloc, and XBox Music, a music service, which is around 160kloc. Since its announcement

⋆ This work was done at Microsoft Research, Cambridge

[3] http://blogs.msdn.com/b/typescript/

in late 2012, the compiler has also been used outside Microsoft, and it is open-source.

The TypeScript type system comprises a number of advanced constructs and concepts. These include structural type equivalence (rather than by-name type equivalence), types for object-based programming (as in object calculi), gradual typing (in the style of Siek and Taha [14]), subtyping of recursive types, and type operators. Collectively, these features should contribute greatly to a harmonious programming experience. One may wonder, still, how they can be made to fit with common JavaScript idioms and codebases. We regard the resolution of this question as one of the main themes in the design of TypeScript.

Interestingly, the designers of TypeScript made a conscious decision not to insist on static soundness. In other words, it is possible for a program, even one with abundant type annotations, to pass the TypeScript typechecker but to fail at run-time with a dynamic type error—generally a trapped error in ordinary JavaScript execution environments. This decision stems from the widespread usage of TypeScript to ascribe types to existing JavaScript libraries and codebases, not just code written from scratch in TypeScript. It is crucial to the usability of the language that it allows for common patterns in popular APIs, even if that means embracing unsoundness in specific places.

The TypeScript language is defined in a careful, clear, but informal document [11]. Naturally, this document contains certain ambiguities. For example, the language permits subtyping recursive types; the literature contains several rules for subtyping recursive types, not all sound, and the document does not say exactly which is employed. Therefore, it may be difficult to know exactly what is the type system, and in what ways it is sound or unsound.

Nevertheless, the world of unsoundness is not a shapeless, unintelligible mess, and unsound languages are not all equally bad (nor all equally good). In classical logic, any two inconsistent theories are equivalent. In programming, on the other hand, unsoundness can arise from a great variety of sins (and virtues). At a minimum, we may wish to distinguish blunders from thoughtful compromises—many language designers and compiler writers are capable of both.

The goal of this paper is to describe the essence of TypeScript by giving a precise definition of its type system on a core set of constructs of the language. This definition clarifies ambiguities of the informal language documentation. It has led to the discovery of a number of unintended inconsistencies and mistakes both in the language specification and in the compiler, which we have reported to the TypeScript team; fortunately, these have been relatively minor and easy to correct. It also helps distinguish sound and unsound aspects of the type system: it provides a basis for partial soundness theorems, and it isolates and explains the sources of unsoundness.

Specifically, in this paper, we identify various core calculi, define precisely their typing rules and, where possible, prove properties of these rules, or discuss why we cannot. The calculi correspond precisely to TypeScript in that every valid program in a given calculus is literally an executable TypeScript program. Since our work took place before the release of TypeScript 1.0, we based it on earlier

versions, in particular TypeScript 0.9.5, which is almost identical to TypeScript 1.0 in most respects; the main differences concern generics. As the design of generics evolved until quite recently, in this paper we restrict attention to the non-generic fragment. Fortunately, for the most part, generics are an orthogonal extension.

The rest of the paper is organized as follows: In §2 we give an informal overview of the design goals of TypeScript. In §3 we give the syntax for a core, featherweight calculus, FTS. In §4 we define safeFTS, a safe, featherweight fragment of TypeScript, by giving details of a type system. In §5 we give an operational semantics for FTS and show how safeFTS satisfies a type soundness property. In §6 we extend the type system of safeFTS obtaining a calculus we refer to as 'production' FTS, or prodFTS for short. This calculus should be thought of as the featherweight fragment of the full TypeScript language, so it is not statically type sound, by design. We characterize the unsound extensions to help understand why the language designers added them. In §7 we give an alternative formulation of the assignment compatibility relation for prodFTS that is analogous to the consistent-subtyping relation of Siek and Taha [14]. We are able to prove that this relation is equal to our original assignment compatibility relation. We briefly review related work in §8 and conclude in §9.

## 2 The Design of TypeScript

The primary goal of TypeScript is to give a statically typed experience to JavaScript development. A syntactic superset of JavaScript, it adds syntax for declaring and expressing types, for annotating properties, variables, parameters and return values with types, and for asserting the type of an expression. This paper's main aim is to formalize these type-system extensions.

TypeScript also adds a number of new language constructs, such as classes, modules, and lambda expressions. The TypeScript compiler implements these constructs by translation to JavaScript (EcmaScript 5). However, these constructs are essentially back-ports of upcoming (EcmaScript 6) JavaScript features and, although they interact meaningfully with the type system, they do not affect its fundamental characteristics.

The intention of TypeScript is not to be a new programming language in its own right, but to enhance and support JavaScript development. Accordingly, a key design goal of the type system is to support current JavaScript styles and idioms, and to be applicable to the vast majority of the many existing—and very popular—JavaScript libraries. This goal leads to a number of distinctive properties of the type system:

**Full erasure:** The types of a TypeScript program leave no trace in the JavaScript emitted by the compiler. There are no run-time representations of types, and hence no run-time type checking. Current dynamic techniques for "type checking" in JavaScript programs, such as checking for the presence of certain properties, or the values of certain strings, may not be perfect, but good enough.

**Structural types:** The TypeScript type system is structural rather than nominal. Whilst structural type systems are common in formal descriptions of object-oriented languages [1], most industrial mainstream languages, such as Java and C♯, are nominal. However, structural typing may be the only reasonable fit for JavaScript programming, where objects are often built from scratch (not from classes), and used purely based on their expected shape.

**Unified object types:** In JavaScript, objects, functions, constructors, and arrays are not separate kinds of values: a given object can simultaneously play several of these roles. Therefore, object types in TypeScript can not only describe members but also contain call, constructor, and indexing signatures, describing the different ways the object can be used. In Featherweight TypeScript, for simplicity, we include only call signatures; constructor and index signatures are broadly similar.

**Type inference:** TypeScript relies on type inference in order to minimize the number of type annotations that programmers need to provide explicitly. JavaScript is a pretty terse language, and the logic shouldn't be obscured by excessive new syntax. In practice, often only a small number of type annotations need to be given to allow the compiler to infer meaningful type signatures.

**Gradual typing:** TypeScript is an example of a gradual type system [14], where parts of a program are statically typed, and others dynamically typed through the use of a distinguished dynamic type, written `any`. Gradual typing is typically implemented using run-time casts, but that is not practical in TypeScript, because of type erasure. As a result, typing errors not identified statically may remain undetected at run-time.

The last point is particularly interesting: it follows from the view that an unsound type system can still be extremely useful. The significant initial uptake of TypeScript certainly suggests that this is the case. While the type system can be wrong about the shape of run-time structures, the experience thus far indicates that it usually won't be. The type system may not be good enough for applications that require precise guarantees (e.g., as a basis for performance optimizations, or for security), but it is more than adequate for finding and preventing many bugs, and, as importantly, for powering a comprehensive and reliable tooling experience of auto-completion, hover tips, navigation, exploration, and refactoring.

In addition to gradual typing, a few other design decisions deliberately lead to type holes and contribute to the unsoundness of the TypeScript type system.

**Downcasting:** The ability to explicitly downcast expressions is common in most typed object-oriented languages. However, in these languages, a downcast is compiled to a dynamic check. In TypeScript, this is not the case, as no trace of the type system is left in the emitted code. So incorrect downcasts are not detected, and may lead to (trapped) run-time errors.

**Covariance:** TypeScript allows unsafe covariance of property types (despite their mutability) and parameter types (in addition to the contravariance that

is the safe choice). Given the ridicule that other languages have endured for this decision, it may seem like an odd choice, but there are significant and sensible JavaScript patterns that just cannot be typed without covariance.

**Indexing:** A peculiar fact of JavaScript is that member access through dot notation is just syntactic sugar for indexing with the member name as a string. Full TypeScript permits specifying indexing signatures, but (in their absence) allows indexing with any string. If the string is a literal that corresponds to a property known to the type system, then the result will have the type of that member (as usual with the dot notation). On the other hand, if the string is not a literal, or does not correspond to a known member, then the access is still allowed, and typed as `any`. Again, this aspect of TypeScript corresponds to common JavaScript usage, and results in another hole in the type system.

One further source of unsoundness may be the treatment of recursive definitions of generic type operators. Deciding type equivalence and subtyping in a structural type system with such definitions is notoriously difficult. Some versions of these problems are equivalent to the equivalence problem for deterministic pushdown automata [15], which was proved decidable relatively recently [13], and which remains a challenging research subject. We do not discuss these points further because we focus on the non-generic fragment of TypeScript, as explained above.

## 3 Featherweight TypeScript

In this section we define the syntax of a core calculus, Featherweight TypeScript (FTS). As mentioned in the introduction, this core calculus covers the non-generic part of TypeScript. To elucidate the design of TypeScript we will refactor the type system into two parts, which we then add to FTS and consider the results as two separate calculi: a 'safe' calculus containing none of the type holes, safeFTS and a complete, 'production' calculus, prodFTS.

Analogously to Featherweight Java [10], our calculi are small and there is a direct correspondence between our calculi and TypeScript: every safeFTS and prodFTS program is literally an executable TypeScript program. (We also make extensive use of the Featherweight Java 'overbar' notation.) However, our calculi are considerably more expressive than Featherweight Java as we retain many impure features that we view as essential to TypeScript programming, such as assignments, variables, and statements.

In this section we define the syntax of our core calculus. The safeFTS type system is defined in §4 and the prodFTS type system is defined in §6.

**FTS expressions:**

| | |
|---|---|
| `e, f` ::= | Expressions |
| `x` | Identifier |
| `l` | Literal |
| `{ ā }` | Object literal |

| | |
|---|---|
| `e=f` | Assignment operator |
| `e` $\oplus$ `f` | Binary operator |
| `e.n` | Property access |
| `e[f]` | Computed property access |
| `e(`$\bar{\texttt{f}}$`)` | Function call |
| `<T>e` | Type assertion |
| `function c { `$\bar{\texttt{s}}$` }` | Function expression |
| `a` ::= `n: e` | Property assignment |
| `c` ::= | Call signature |
| `(`$\bar{\texttt{p}}$`)` | Parameter list |
| `(`$\bar{\texttt{p}}$`): T` | Parameter list with return type |
| `p` ::= | Parameter |
| `x` | Identifier |
| `x:T` | Typed identifier |

As TypeScript includes JavaScript as a sublanguage, thus Featherweight TypeScript contains what can be thought of as Featherweight JavaScript. We highlight in grey the constructs that are new to TypeScript and not part of JavaScript.

FTS expressions include literals, `l`, which can be a number $n$, a string $s$, or one of the constants `true`, `false`, `null`, or `undefined`.[4] We assume a number of built-in binary operators, such as `===`, `>`, `<`, and `+`. In the grammar we use $\oplus$ to range over all the binary operators, and do not specify them further as their meaning is clear. We assume that `x`, `y`, and `z` range over valid identifiers and `n` ranges over property names. We also assume that the set of identifiers includes the distinguished identifier `this` which cannot be used as a formal parameter or declared as a local.

FTS supports both property access and computed property access. Function expressions extend those of JavaScript by optionally including parameter and return type annotations on call signatures. (TypeScript also features a more compact 'arrow' form for function expressions; for example one can write `(x) => x+1` instead of the more verbose `function (x) { return x + 1; }`.)

**FTS statements:**

| | |
|---|---|
| `s, t` ::= | Statement |
| `e;` | Expression statement |
| `if (e) {`$\bar{\texttt{s}}$`} else {`$\bar{\texttt{t}}$`}` | If statement |
| `return;` | Return statement |
| `return e;` | Return value statement |
| `v;` | Variable statement |
| `u, v` ::= | Variable declaration |
| `var x:T` | Uninitialized typed variable declaration |
| `var x:T = e` | Initialized typed variable declaration |
| `var x` | Uninitialized variable declaration |
| `var x = e` | Initialized variable declaration |

[4] JavaScript somewhat confusingly supports two primitive values: `null` (an object) and `undefined` which, for example, is returned when accessing a non-existent property.

For the sake of compactness, we support conditional statements but not conditional expressions. Variable declarations are extended from JavaScript to include optional type annotations.

**FTS types:**

| | |
|---|---|
| `R, S, T ::=` | Type |
| `any` | Any type |
| `P` | Primitive type |
| `O` | Object type |
| `P ::=` | Primitive type |
| `number` | Number |
| `string` | String |
| `boolean` | Boolean type |
| `void` | Void type |
| *`Null`* | Null type |
| *`Undefined`* | Undefined type |
| `O ::=` | Object type |
| `I` | Interface type |
| `L` | Literal type |
| `L ::= { M̄ }` | Object type literal |
| `M, N ::=` | Type member |
| `n:T` | Property |
| `(x̄: S̄): T` | Call signature |

FTS types fall into three categories: primitive types, object types, and a distinguished type, written `any`.

The primitive types include the run-time primitive types of JavaScript: `number` for 64 bit IEEE 754 floating point numbers, `string` for Unicode UTF-16 strings, and `boolean` to denote the boolean values. The `void` type denotes an absence of a value, which arises from running a function that simply returns without giving a result. There are no values of this type. There are two further types, *`Null`* and *`Undefined`*, that are expressible but not denotable; we write them in italics to further emphasize their special status. In other words, these two types cannot be referenced in valid TypeScript programs, but they arise within the typing process.

FTS object types consist of interface types and literal types. For compactness, we do not support classes in FTS. At the level of the type system, classes are secondary, and do not add any significant new issues, but complicate the formalization of the language and the operational semantics. For that reason we omit them, but do keep interfaces. Similarly we drop array, function, and constructor type literals. FTS supports object type literals, whose type members can include properties and call signatures. The inclusion of call signature properties enable us to encode function literal types; for example the type `(x:S) => T` can be encoded as the type `{(x:S): T}`. We refer to an object type literal that contains a call signature as a *callable* type, and we assume a predicate *callable* that returns

true if the type contains a call signature.[5] It is important to note that the type `{ }` (i.e., the empty object type literal) is a valid type. In addition, TypeScript has a number of predefined interfaces that are always in scope in TypeScript programs. For the purposes of this paper, these interfaces are `Object`, `Function`, `String`, `Number`, and `Boolean`.

**FTS declaration:**

| | |
|---|---|
| `D` ::= | Interface declaration |
| `interface I { M̄ }` | |
| `interface I extends Ī { M̄ }` | (Ī non-empty) |

FTS supports only one form of declaration: an interface. An interface allows a name to be associated with an object type. Thus, a declaration `interface I { M̄ }` associates with the name `I` the object type literal `{ M̄ }`. However, a couple of subtleties arise. First, interfaces can be recursive; and indeed a collection of interface declarations can be mutually recursive. Also, interfaces can inherit from zero or more base types (which, in the case of FTS must be interfaces). In this case an interface has all the members defined in its immediate declaration and furthermore all the members of the base types.

In TypeScript this process of inheritance is further complicated by the notion of an interface hiding members of its base types, but for FTS we shall make the simplifying assumption that no hiding is possible. We do not model the notion of private members; in FTS all members are public.

An interface table $\Sigma$ is a map from an interface name `I` to an interface declaration `D`. A program is then a pair $(\Sigma, \bar{\mathtt{s}})$ of an interface table and a sequence of statements. In order to reduce notational overload, we assume a single fixed interface table $\Sigma$. The interface table induces relationships between types (subtyping and assignment compatibility); these relations are defined in later sections.

The given interface table must satisfy some familiar sanity conditions:

1. $\Sigma(\mathtt{I}) = \mathtt{interface\ I} \ldots$ for every $\mathtt{I} \in dom(\Sigma)$;
2. for every interface name `I` appearing anywhere in $\Sigma$, it is the case that $\mathtt{I} \in dom(\Sigma)$; and
3. there are no cycles in the dependency graph induced by the `extends` clauses of the interface declarations defined in $\Sigma$.

This last point rules out declarations such as the following:

```
// Error: Self-cyclic extends clause
interface I extends I { ... }

// Error: Cyclic extends clauses
interface J extends K { ... }
interface K extends J { ... }
```

[5] In FTS we do not support functions with multiple call signatures and thus we ignore the process of overloading resolution in TypeScript.

Throughout the rest of the paper, we write $\Gamma$ to denote a type environment, which is a function from identifiers to types. We write $\Gamma, \texttt{x}\colon \texttt{T}$ to denote the extension of the type environment $\Gamma$ with the mapping of identifier `x` to type `T`. This extension is defined only if $\texttt{x} \notin \mathit{dom}(\Gamma)$. In some cases we need to override a function mapping; we write $\Gamma \uplus \texttt{x}\colon \texttt{T}$ to denote the function that maps `x` to `T`, and otherwise maps an identifier $\texttt{y} \neq \texttt{x}$ to $\Gamma(\texttt{y})$.

## 4 Safe Featherweight TypeScript (safeFTS)

In this section we define the safeFTS calculus which adds a type system to the FTS calculus defined in the previous section. As suggested by its name, this type system, although a subsystem of the full TypeScript type system, has familiar safety properties. (These properties are treated in §5.)

Our first step is to define an important type relation in TypeScript: assignment compatibility [11, §3.8.3]. This relation is written $\texttt{S} \leqq \texttt{T}$ and captures the intuition that a value of type `S` can be assigned to a value of type `T`. However, the presence of interfaces immediately makes this relation a little tricky to define. For example, consider the following interface declaration.

```
interface I {
   a : number,
   (x: string): I
}
```

As TypeScript has a structural type system, this actually defines a type `I` which is described by the following *equation*.

`I` = `{ a : number, (x: string): I }`

A value of type `I` is an object with a property `a` of type `number`, and a function that maps strings to values of type `I`. Clearly this is equivalent to an object with a property `a` of type `number`, and a function that maps strings to objects with a property `a` of type `number`, and a function that maps strings to values of type `I`, and so on, *ad infinitum*. The language specification notes this potential infinite expansion [11, §3.8.1] but gives few details about how it is to be dealt with. (Indeed, the discussion of types excludes any mention of interface names, which are assumed to have been replaced by their definitions.)

Fortunately, this equi-recursive treatment of recursive types has a pleasant, but slightly less well-known formalization that views types as (finite or infinite) trees, uses greatest fixed points to define type relationships, and coinduction as a proof technique. We give only the basic definitions but the excellent survey article [6] offers further details.

We represent types as possibly infinite trees, with nodes labelled by a symbol from the set $O = \{\texttt{any}, \texttt{null}, \texttt{undefined}, \texttt{boolean}, \texttt{number}, \texttt{string}, \texttt{\{\}}, \rightarrow\}$. The branches are labelled with a name taken from the set $B = \texttt{X} \cup \texttt{N} \cup \{\texttt{ret}, \texttt{cs}\}$, where `X` is the set of FTS identifiers, `N` is the set of FTS property names, and $\texttt{ret} \notin \texttt{X}$ and $\texttt{cs} \notin \texttt{N}$ are distinguished names (used to signify a return type and a call signature). We write $B^\star$ for the set of sequences of elements $b \in B$. The

empty sequence is written $\bullet$, and if $\pi$ and $\pi'$ are sequences, then we write $\pi \cdot \pi'$ for the concatenation of $\pi$ and $\pi'$.

**Definition 1.** *A tree type is a partial function* $\mathcal{T}\colon B^\star \rightharpoonup O$ *such that:*

- $\mathcal{T}(\bullet)$ *is defined.*
- *If* $\mathcal{T}(\pi \cdot \sigma)$ *is defined then* $\mathcal{T}(\pi)$ *is defined.*
- *If* $\mathcal{T}(\pi) = \{\}$ *then* $\exists P \subseteq (M \cup \{\mathtt{cs}\}).P = \{n_1, \ldots, n_p\}$ *such that* $\mathcal{T}(\pi \cdot n_1), \ldots, \mathcal{T}(\pi \cdot n_p)$ *are defined and* $\forall b \in B.b \notin P$ *implies* $\mathcal{T}(\pi \cdot b)$ *is undefined.*
- *If* $\mathcal{T}(\pi) = \rightarrow$ *then* $\exists X \subseteq X.X = \{x_1, \ldots, x_p\}$ *such that* $\mathcal{T}(\pi \cdot x_1), \ldots, \mathcal{T}(\pi \cdot x_p)$ *and* $\mathcal{T}(\pi \cdot \mathtt{ret})$ *are defined and* $\forall b \in B.b \notin X$ *implies* $\mathcal{T}(\pi \cdot b)$ *is undefined.*
- *If* $\mathcal{T}(\pi) \in \{\mathtt{any}, \mathtt{Null}, \mathtt{Undefined}, \mathtt{boolean}, \mathtt{number}, \mathtt{string}\}$ *then* $\forall b \in B.\mathcal{T}(\pi \cdot b)$ *is undefined.*

The set of all tree types is written $\mathcal{T}$. For notational convenience, we write any for the tree T with $\mathtt{T}(\bullet) = \mathtt{any}$, and likewise for the other nullary type constructors. If $\mathtt{T}_1$ and $\mathtt{T}_2$ are types, then we write $\{\mathtt{n}_1\mathtt{:}\mathtt{T}_1\mathtt{,}\mathtt{n}_2\mathtt{:}\mathtt{T}_2\}$ for the tree type T such that $\mathtt{T}(\bullet) = \{\}$, $\mathtt{T}(\mathtt{n}_1) = \mathtt{T}_1$, and $\mathtt{T}(\mathtt{n}_2) = \mathtt{T}_2$. Similarly, if $\mathtt{T}_1$ and $\mathtt{T}_2$ are types, then we write $\{(\mathtt{x}\mathtt{:}\mathtt{T}_1)\mathtt{:}\ \mathtt{T}_2\}$ for the tree type T such that $\mathtt{T}(\bullet) = \{\}$, $\mathtt{T}(\mathtt{cs}) = \rightarrow$, $\mathtt{T}(\mathtt{cs} \cdot \mathtt{x}) = \mathtt{T}_1$ and $\mathtt{T}(\mathtt{cs} \cdot \mathtt{ret}) = \mathtt{T}_2$. We restrict our attention to finitely branching trees, but trees may well still be infinite.

**Definition 2.** *Two tree types* $\mathcal{S}$ *and* $\mathcal{T}$ *are assignment compatible if the pair* $(\mathcal{S}, \mathcal{T})$ *is in the greatest fixed point of the following function* $\mathcal{A}\colon \mathcal{P}(\mathcal{T} \times \mathcal{T}) \to \mathcal{P}(\mathcal{T} \times \mathcal{T})$.

$$
\begin{aligned}
\mathcal{A}(R) = {} & \{(\mathcal{S}, \mathcal{S}) \mid \mathcal{S} \vdash \diamond\} \cup \{(\mathcal{S}, \mathtt{any}) \mid \mathcal{S} \vdash \diamond\} \cup \{(\mathtt{Undefined}, \mathcal{T}) \mid \mathcal{T} \vdash \diamond\} \\
& \cup \{(\mathtt{Null}, \mathcal{T}) \mid \mathcal{T} \vdash \diamond \text{ and } \mathcal{T} \neq \mathtt{Undefined}\} \cup \{(P, \mathcal{T}) \mid (\mathcal{I}(P), \mathcal{T}) \in R\} \\
& \cup \{(\{\ \bar{M}_0\mathtt{,}\ \bar{M}_1\ \}, \{\ \bar{M}_2\ \}) \mid \{\ \bar{M}_0\mathtt{,}\ \bar{M}_1\ \} \vdash \diamond \text{ and } \bar{M}_1 \sim \bar{M}_2\} \\
& \quad \textit{where } n_1\mathtt{:}\mathcal{T}_1 \sim n_2\mathtt{:}\mathcal{T}_2 \textit{ if } n_1 \equiv n_2 \textit{ and } \mathcal{T}_1 \equiv \mathcal{T}_2 \\
& \qquad (\bar{x}\mathtt{:}\bar{S})\mathtt{:}R_0 \sim (\bar{y}\mathtt{:}\bar{T})\mathtt{:}R_1 \textit{ if } (\bar{T}, \bar{S}) \in R, R_1 \neq \mathtt{void} \textit{ and } (R_0, R_1) \in R \\
& \qquad (\bar{x}\mathtt{:}\bar{S})\mathtt{:}R \sim (\bar{y}\mathtt{:}\bar{T})\mathtt{:}\mathtt{void} \textit{ if } (\bar{T}, \bar{S}) \in R
\end{aligned}
$$

In this definition we make use of a wellformedness predicate on types, written $\mathtt{S} \vdash \diamond$, whose simple definition we omit for lack of space. We also make use of a helper function, $\mathcal{I}$, to replace a primitive type (boolean, number, string) with its associated interface type (Boolean, Number, String, respectively).

We can also define assignment compatibility using a familiar collection of inference rules, but it should be noted this is a *coinductively* defined relation. We use double horizontal lines to emphasize this distinction.

**safeFTS assignment compatibility:** $\mathtt{S} \leqq \mathtt{T}$ **and** $\mathtt{M}_0 \leqq \mathtt{M}_1$

$$\frac{\mathtt{S} \vdash \diamond}{\mathtt{S} \leqq \mathtt{S}}\ [\text{A-Refl}] \qquad \frac{\mathtt{S} \vdash \diamond}{\mathtt{S} \leqq \mathtt{any}}\ [\text{A-AnyR}] \qquad \frac{\mathtt{T} \vdash \diamond}{\mathtt{Undefined} \leqq \mathtt{T}}\ [\text{A-Undef}]$$

$$\frac{\mathtt{T} \vdash \diamond \qquad \mathtt{T} \neq \mathtt{Undefined}}{\mathtt{Null} \leqq \mathtt{T}}\ [\text{A-Null}] \qquad \frac{\mathcal{I}(\mathtt{P}) \leqq \mathtt{T}}{\mathtt{P} \leqq \mathtt{T}}\ [\text{A-Prim}]$$

$$\frac{\{\ \bar{M}_0, \bar{M}_1\ \} \vdash \diamond \qquad \bar{M}_1 \leqq \bar{M}_2}{\{\ \bar{M}_0, \bar{M}_1\ \} \leqq \{\ \bar{M}_2\ \}}\ \text{[A-Object]} \qquad \mathtt{n}:\mathtt{T} \leqq \mathtt{n}:\mathtt{T}\ \text{[A-Prop]}$$

$$\frac{\bar{T} \leqq \bar{S} \qquad R_1 \neq \mathtt{void} \qquad R_0 \leqq R_1}{(\bar{x}:\bar{S}):R_0 \leqq (\bar{y}:\bar{T}):R_1}\ \text{[A-CS]} \qquad \frac{\bar{T} \leqq \bar{S} \qquad R \vdash \diamond}{(\bar{x}:\bar{S}):R \leqq (\bar{y}:\bar{T}):\mathtt{void}}\ \text{[A-CS-Void]}$$

Rule [A-Refl] states that any type can be assigned to itself, and rule [A-AnyR] that any type can be assigned to `any`. In rule [A-Undef] the type *`Undefined`* can be assigned to any type; and in rule [A-Null] the type *`Null`* can be assigned to any type except *`Undefined`*. The effect of these rules is that, when viewing assignment compatibility as an order, *`Undefined`* is the least type, *`Null`* is below any user-defined type, and that all types are below `any`, which is the top type. Rule [A-Prop] states that assignment compatibility is *invariant* on property members, and rules [A-CS] and [A-CS-Void] capture the fact that assignment compatibility is contra-/co-variant on call signatures.

Note that there is no explicit transitivity rule for assignment compatibility, but for safeFTS it is derivable.

**Lemma 1 (Transitivity derived rule).**

1. *If* $S \leqq T$ *and* $T \leqq U$ *then* $S \leqq U$
2. *If* $M_0 \leqq M_1$ *and* $M_1 \leqq M_2$ *then* $M_0 \leqq M_2$

The proof of this lemma is analogous to that of Gapeyev et al.'s Theorem 4.7 [6].

The type system for TypeScript, and hence safeFTS, consists of two inter-defined typing relations: one where type information is inferred and one where some type context is taken into account when a type is inferred. In this respect, TypeScript is reminiscent of local type inference systems [12], but the detail is different. The first relation is written $\Gamma \vdash \mathtt{e}:\mathtt{T}$ and is read "given type environment $\Gamma$, the expression `e` has type `T`." The second relation, written $\Gamma \vdash \mathtt{e} \downarrow \mathtt{S}:\mathtt{T}$, is read "given type environment $\Gamma$, the expression `e` *in the context of type* `S` has type `T`." This relation is called 'contextual typing' in the language specification [11, §4.18].

**Expression typing:** $\Gamma \vdash \mathtt{e}:\mathtt{T}$

$$\text{[I-Id]}\ \frac{}{\Gamma, \mathtt{x}:\mathtt{T} \vdash \mathtt{x}:\mathtt{T}} \qquad \text{[I-Number]}\ \frac{}{\Gamma \vdash n:\mathtt{number}}$$

$$\text{[I-String]}\ \frac{}{\Gamma \vdash s:\mathtt{string}} \qquad \text{[I-Bool]}\ \frac{}{\Gamma \vdash \mathtt{true}, \mathtt{false}:\mathtt{boolean}}$$

$$\text{[I-Null]}\ \frac{}{\Gamma \vdash \mathtt{null}:\mathit{Null}} \qquad \text{[I-Undefined]}\ \frac{}{\Gamma \vdash \mathtt{undefined}:\mathit{Undefined}}$$

$$\text{[I-ObLit]}\ \frac{\Gamma \vdash \bar{\mathtt{e}}:\bar{\mathtt{T}}}{\Gamma \vdash \{\ \bar{\mathtt{n}}:\bar{\mathtt{e}}\ \}:\{\ \bar{\mathtt{n}}:\bar{\mathtt{T}}\ \}} \qquad \text{[I-Assign]}\ \frac{\Gamma \vdash \mathtt{e}:\mathtt{S} \qquad \Gamma \vdash \mathtt{f} \downarrow \mathtt{S}:\mathtt{T} \qquad \mathtt{T} \leqq \mathtt{S}}{\Gamma \vdash \mathtt{e} = \mathtt{f}:\mathtt{T}}$$

$$\text{[I-Op]}\ \frac{\Gamma \vdash \mathtt{e}:\mathtt{S}_0 \qquad \Gamma \vdash \mathtt{f}:\mathtt{S}_1 \qquad \mathtt{S}_0 \oplus \mathtt{S}_1 = \mathtt{T}}{\Gamma \vdash \mathtt{e} \oplus \mathtt{f}:\mathtt{T}}$$

$$\text{[I-Prop]}\ \frac{\Gamma \vdash \texttt{e}: \texttt{S} \qquad \mathit{lookup}(\texttt{S}, \texttt{n}) = \texttt{T}}{\Gamma \vdash \texttt{e.n}: \texttt{T}}$$

$$\text{[I-CompProp]}\ \frac{\Gamma \vdash \texttt{e}: \texttt{S} \qquad \texttt{S} \leqq \texttt{Object} \qquad \Gamma \vdash \texttt{f}: \texttt{string}}{\Gamma \vdash \texttt{e[f]}: \texttt{any}}$$

$$\text{[I-Call]}\ \frac{\Gamma \vdash \texttt{e}: \texttt{\{ (}\bar{\texttt{x}}\texttt{: }\bar{\texttt{S}}\texttt{): R \}} \qquad \Gamma \vdash \bar{\texttt{f}} \downarrow \bar{\texttt{S}}: \bar{\texttt{T}} \qquad \bar{\texttt{T}} \leqq \bar{\texttt{S}}}{\Gamma \vdash \texttt{e(}\bar{\texttt{f}}\texttt{)}: \texttt{R}}$$

$$\text{[I-Assert]}\ \frac{\Gamma \vdash \texttt{e}: \texttt{S} \qquad \texttt{S} \leqq \texttt{T}}{\Gamma \vdash \texttt{<T>e}: \texttt{T}}$$

$$\text{[I-Func1]}\ \frac{\Gamma_1, \texttt{this}: \texttt{any}, |\bar{\texttt{p}}| \vdash \mathit{getVars}(\bar{\texttt{s}}) \rightsquigarrow \Gamma_2 \qquad \Gamma_2 \vdash \bar{\texttt{s}} \downarrow \texttt{T}: \bar{\texttt{R}}}{\Gamma_1 \vdash \texttt{function (}\bar{\texttt{p}}\texttt{): T \{ }\bar{\texttt{s}}\texttt{ \}: \{ (}|\bar{\texttt{p}}|\texttt{): }\mathit{return}(\bar{\texttt{R}})\texttt{ \}}}$$

$$\text{[I-Func2]}\ \frac{\Gamma_1, \texttt{this}: \texttt{any}, |\bar{\texttt{p}}| \vdash \mathit{getVars}(\bar{\texttt{s}}) \rightsquigarrow \Gamma_2 \qquad \Gamma_2 \vdash \bar{\texttt{s}}: \bar{\texttt{R}}}{\Gamma_1 \vdash \texttt{function (}\bar{\texttt{p}}\texttt{) \{ }\bar{\texttt{s}}\texttt{ \}: \{ (}|\bar{\texttt{p}}|\texttt{): }\mathit{return}(\bar{\texttt{R}})\texttt{ \}}}$$

On the whole, these rules are routine. In rule [I-Assign], the expression `e = f` has type `T`, if the subexpression `e` has some type `S` and the subexpression `f` *in the context of* `S` has type `T`. We also check that type `T` is assignment compatible with type `S`(for reasons that should become clearer once contextual typing is defined).

In rule [I-Op], when typing the use of a built-in binary operator $\oplus$, we overload notation and use a binary (partial) function $\oplus$ to calculate the return type given the types of the two arguments. Interestingly, the current language specification states that certain combinations of types should be considered both an error *and* yield the return type `any`. The exact details of these type functions [11, §4.15] are omitted from this paper as they are somewhat orthogonal to our main concerns.

Rule [I-Prop] details typing for property access. It makes use of an auxiliary, partial function $\mathit{lookup}(\texttt{S}, \texttt{n})$ that returns the type of property `n`, if it exists, of a type `S`. This process is a little subtle as TypeScript allows primitive types to have properties, and all object types inherit properties from the `Object` interface. The auxiliary function is defined for safeFTS as follows:[6]

$$\mathit{lookup}(\texttt{S}, \texttt{n}) = \begin{cases} \mathit{lookup}(\texttt{Number}, \texttt{n}) & \text{if } \texttt{S} = \texttt{number} \\ \mathit{lookup}(\texttt{Boolean}, \texttt{n}) & \text{if } \texttt{S} = \texttt{boolean} \\ \mathit{lookup}(\texttt{String}, \texttt{n}) & \text{if } \texttt{S} = \texttt{string} \\ \texttt{T} & \text{if } \texttt{S} = \texttt{\{ }\bar{\texttt{M}}_0\texttt{,n:T,}\bar{\texttt{M}}_1\texttt{ \}} \\ \mathit{lookup}(\texttt{Object}, \texttt{n}) & \text{if } \texttt{S} = \texttt{\{ }\bar{\texttt{M}}\texttt{ \}} \text{ and } \texttt{n} \notin \bar{\texttt{M}} \end{cases}$$

In rule [I-CompProp] a computed property expression `e[f]` has type `any` if subexpression `e` has type `S` (which must be assignable to the type `Object`) and subexpression `f` has type `string`.

In rule [I-Call] a function call `e(`$\bar{\texttt{f}}$`)` has type `R` if the subexpression `e` has the call signature type `{ (`$\bar{\texttt{x}}$`: `$\bar{\texttt{S}}$`): R }`. We also check that the arguments $\bar{\texttt{f}}$ in the

[6] TypeScript also allows all callable object types to inherit properties from the `Function` interface.

context of types $\bar{\mathtt{S}}$ have types $\bar{\mathtt{T}}$, and that types $\bar{\mathtt{T}}$ are assignment compatible with types $\bar{\mathtt{S}}$.

In rule [I-Assert] a type assertion `<T>e` has type `T` if subexpression `e` has type `S` where type `S` is assignable to `T`. In safeFTS the only asserts permitted are those that are known to be correct.

Rules [I-Func1] and [I-Func2] address typing a function expression. Both rules assume that the type of `this` in a function expression is `any`. Both rules also make use of an auxiliary function $|\cdot|$ to extract types from the parameters in a call signature. If a parameter does not have a type annotation, then TypeScript assumes that it is of type `any`. One consequence of this design is that TypeScript does not try to infer types for the parameters of a function expression.

```
var fact = function (x) {
    if (x == 0) { return 1; }
    else { return x * fact(x - 1); }
};                    // infers type { (x:any): number }
```

Both rules run into an "awful" feature (using the terminology of Crockford [5]) inherited from JavaScript: all variables declared in a function body are in scope regardless of nesting levels, order, or even how many times they are declared. The whole function body (except for functions nested inside it) is treated as a flat declaration space. In other words, JavaScript does not have block scoping. Thus the following (buggy) JavaScript code:

```
var scope = function (p) {
    var y = 1;
    var a = [y, x, z];
    var x = 2;
    if (test(p)) { var z = 3; }
    else { var z = 4; var w = 5; }
    return w + a[2];
};
```

is treated as if it had instead been written as follows.

```
var scope = function (p) {
    var y = 1;
    var x; var z; var w;                    // implicit
    var a = [y, x, z];
    var x = 2;
    if (test(p)) { var z = 3; }
    else { var z = 4; var w = 5; }
    return w + a[2];
};
```

At the level of typing this means that when typing a function expression, we need two phases: first, we find the types of all the local variables declared in the function body; and second, we then type the function body using the type environment extended with the types determined from the first phase.

There is a further complication as TypeScript also infers types of local variables with initializers. Furthermore, TypeScript, again following JavaScript, supports mutually recursive variable declarations. We assume a function $getVars(\bar{\mathtt{s}})$

that returns the variable declarations in the scope of the sequence of statements $\bar{\mathtt{s}}$. This function needs to deal with the problem of a collection of untyped initialized variable declarations that depend on each other. In the case where such a collection is cyclic, the language specification states that they should all be treated as if they were explicitly typed as `any`. A non-cyclic collection of untyped variable declarations are reordered in reverse dependency order.

Given this sequence of variable declarations, we define a judgement written $\Gamma_1 \vdash \bar{\mathtt{v}} \rightsquigarrow \Gamma_2$ to extend a given type environment $\Gamma_1$ with the type information contained in the variable declarations $\bar{\mathtt{v}}$ yielding a new type environment $\Gamma_2$. The chief concern is dealing with repeated variable declarations. Such repetition is permitted in TypeScript provided that the multiple declarations associate the same type with the variable [11, §5.1].

**Environment extension:** $\Gamma_1 \vdash \bar{\mathtt{v}} \rightsquigarrow \Gamma_2$

$$\frac{}{\Gamma \vdash \bullet \rightsquigarrow \Gamma} \qquad \frac{\mathit{dupOK}(\Gamma_1, \mathtt{x}\colon \mathtt{T}) \qquad \Gamma_1 \uplus (\mathtt{x}\colon \mathtt{T}) \vdash \bar{\mathtt{v}} \rightsquigarrow \Gamma_2}{\Gamma_1 \vdash \mathtt{var}\ \mathtt{x}\colon \mathtt{T};\ \bar{\mathtt{v}} \rightsquigarrow \Gamma_2}$$

$$\frac{\mathit{dupOK}(\Gamma_1, \mathtt{x}\colon \mathtt{any}) \qquad \Gamma_1 \uplus (\mathtt{x}\colon \mathtt{any}) \vdash \bar{\mathtt{v}} \rightsquigarrow \Gamma_2}{\Gamma_1 \vdash \mathtt{var}\ \mathtt{x};\ \bar{\mathtt{v}} \rightsquigarrow \Gamma_2}$$

$$\frac{\mathit{dupOK}(\Gamma_1, \mathtt{x}\colon \mathtt{T}) \qquad \Gamma_1 \uplus (\mathtt{x}\colon \mathtt{T}) \vdash \bar{\mathtt{v}} \rightsquigarrow \Gamma_2}{\Gamma_1 \vdash \mathtt{var}\ \mathtt{x}\colon \mathtt{T} = \mathtt{e};\ \bar{\mathtt{v}} \rightsquigarrow \Gamma_2}$$

$$\frac{\Gamma_1 \vdash \mathtt{e}\colon \mathtt{T} \qquad \mathit{dupOK}(\Gamma_1, \mathtt{x}\colon \mathtt{T}) \qquad \Gamma_1 \uplus (\mathtt{x}\colon \mathtt{T}) \vdash \bar{\mathtt{v}} \rightsquigarrow \Gamma_2}{\Gamma_1 \vdash \mathtt{var}\ \mathtt{x} = \mathtt{e};\ \bar{\mathtt{v}} \rightsquigarrow \Gamma_2}$$

We use the following predicate to detect duplicates:

$$\mathit{dupOK}(\Gamma, \mathtt{x}\colon \mathtt{T}) = \begin{cases} \mathit{true} & \text{if } \mathtt{x} \notin \mathit{dom}(\Gamma) \text{ or } \Gamma(\mathtt{x}) = \mathtt{T} \\ \mathit{false} & \text{otherwise} \end{cases}$$

Returning to the typing rules [I-Func1] and [I-Func2] we use an auxiliary function $\mathit{return}(\bar{\mathtt{R}})$ to calculate the overall return type given the types $\bar{\mathtt{R}}$ inferred from the return statements in the body of the function. This function is defined as follows.

$$\mathit{return}(\bar{\mathtt{R}}) = \begin{cases} \mathtt{void} & \text{if } \bar{\mathtt{R}} = \bullet \\ \mathit{widen}(\mathtt{S}) & \text{if } \mathtt{S} = \mathit{bct}(\bar{\mathtt{R}}) \end{cases}$$

In calculating the return type we make use of two important functions on types. The first function, $\mathit{widen}(\mathtt{T})$, calculates the widened form [11, §3.9] of a type `T`. This is the type `T` with all occurrences of the expressible but not denotable types, ***`Null`*** and ***`Undefined`***, replaced by the type `any`.

The second function $\mathit{bct}(\bar{\mathtt{S}})$ calculates the best common type [11, §3.10] of a sequence of types $\bar{\mathtt{S}}$ and is defined to be a type taken from the sequence $\bar{\mathtt{S}}$ such that all the other types in the sequence can be assigned to it. For example, the best common type of the primitive type `number` and the empty object type `{ }`

is the empty object type; whereas the types `number` and `string` have no best common type.

In a small number of situations, more precision can be gained by using explicit type information when typing expressions. For example, in TypeScript the expression `function(s) { return s.length; }` has type `{(s: any): any}`. But, in the context of the explicitly typed declaration

```
var f: (s:string) => number;
```

within the assignment expression `f = function(s) { return s.length; }` we should type the function knowing that the parameter `s` has the type `string`. Moreover, the information flow can be more than one-way. Thus given the declaration

```
var g: (s:string) => any;
```

the assignment expression `g = function(s) { return s.length; }` actually has the type `(s:string) => number`.

As mentioned earlier, the contextual typing relation is written $\Gamma \vdash \mathtt{e} \downarrow \mathtt{S} : \mathtt{T}$, and defined as follows.

**Expression contextual typing $\Gamma \vdash \mathtt{e} \downarrow \mathtt{S} : \mathtt{T}$ and $\Gamma \vdash \mathtt{a} \downarrow \mathtt{L} : \mathtt{M}$**

$$[\text{C-ObLit}]\ \frac{\Gamma \vdash \bar{\mathtt{a}} \downarrow \mathtt{L} : \bar{\mathtt{M}}}{\Gamma \vdash \mathtt{\{}\ \bar{\mathtt{a}}\ \mathtt{\}} \downarrow \mathtt{L} : \mathtt{\{}\ \bar{\mathtt{M}}\ \mathtt{\}}}$$

$$[\text{C-PA1}]\ \frac{(\mathtt{x}\mathtt{:}\ \mathtt{S}) \in \bar{\mathtt{M}} \quad \Gamma \vdash \mathtt{e} \downarrow \mathtt{S} : \mathtt{T}}{\Gamma \vdash (\mathtt{x}\mathtt{:}\ \mathtt{e}) \downarrow \mathtt{\{}\ \bar{\mathtt{M}}\ \mathtt{\}} : (\mathtt{x}\mathtt{:}\ \mathtt{T})} \qquad [\text{C-PA2}]\ \frac{(\mathtt{x}\mathtt{:}\ \mathtt{S}) \notin \bar{\mathtt{M}} \quad \Gamma \vdash \mathtt{e} : \mathtt{T}}{\Gamma \vdash (\mathtt{x}\mathtt{:}\ \mathtt{e}) \downarrow \mathtt{\{}\ \bar{\mathtt{M}}\ \mathtt{\}} : (\mathtt{x}\mathtt{:}\ \mathtt{T})}$$

$$[\text{C-Func}]\ \frac{\Gamma, \bar{\mathtt{x}} : \bar{\mathtt{S}}, \mathtt{this} : \mathtt{any} \vdash \bar{\mathtt{s}} \downarrow \mathtt{T} : \bar{\mathtt{R}} \quad \mathtt{R} = \mathit{return}(\bar{\mathtt{R}})}{\Gamma \vdash \mathtt{function}\ (\bar{\mathtt{x}})\ \mathtt{\{}\bar{\mathtt{s}}\mathtt{\}} \downarrow \mathtt{\{}\ (\bar{\mathtt{y}}\mathtt{:}\ \bar{\mathtt{S}})\mathtt{:}\ \mathtt{T}\ \mathtt{\}} : \mathtt{\{}\ (\bar{\mathtt{y}}\mathtt{:}\ \bar{\mathtt{S}})\mathtt{:}\ \mathtt{R}\ \mathtt{\}}}$$

$$[\text{C-Inf}]\ \frac{\Gamma \vdash \mathtt{e} : \mathtt{T}}{\Gamma \vdash \mathtt{e} \downarrow \mathtt{S} : \mathtt{T}}$$

In rule [C-ObLit], in order to contextually type the the object literal $\mathtt{\{}\ \bar{\mathtt{a}}\ \mathtt{\}}$, we contextually type the property assignments $\bar{\mathtt{a}}$. In rule [C-PA1] the property assignment $\mathtt{x}\mathtt{:}\ \mathtt{e}$ in the context of the object type literal $\mathtt{\{}\ \bar{\mathtt{M}}\ \mathtt{\}}$ (which supports property `x` at type `S`) has type $\mathtt{x}\mathtt{:}\ \mathtt{T}$ where the subexpression `e` has type `T` in the context of type `S`. Rule [C-PA2] covers the case where the contextual type does not support the property `x`. In this case the type is inferred from the subexpression `e`.

In rule [C-Func] the function expression $\mathtt{function}\ (\bar{\mathtt{x}})\ \mathtt{\{}\bar{\mathtt{s}}\mathtt{\}}$ in the context of the type $\mathtt{\{}\ (\bar{\mathtt{y}}\mathtt{:}\ \bar{\mathtt{S}})\mathtt{:}\ \mathtt{T}\ \mathtt{\}}$ (where the length of the sequences $\bar{\mathtt{x}}, \bar{\mathtt{y}}$ and $\bar{\mathtt{S}}$ are equal) has the type $\mathtt{\{}\ (\bar{\mathtt{y}}\mathtt{:}\ \bar{\mathtt{S}})\mathtt{:}\ \mathtt{R}\ \mathtt{\}}$ if the function body $\bar{\mathtt{s}}$ has the types $\bar{\mathtt{R}}$ in the context of type $\bar{\mathtt{T}}$ and `R` is the result of the calculating the return type from the sequence of types $\bar{\mathtt{R}}$. Rule [C-Inf] applies only if the expression `e` is not a function expression or an object literal, and asserts that expression `e` in the context of type `S` has type `T` simply if `e` has type `T`; the contextual type is ignored.

Thus contextual typing is highly (and to the authors' minds, uncomfortably) syntax dependent. For example, a misplaced pair of brackets can affect the contextual typing of a TypeScript expression [11, §4.18].

```
var t1: (s: string) => any;
var t2 = (t1 = function (s) { return s.length; });
                    // Contextual typing! Infers { (s: string): number }
var t3 = (t1 = (function (s) { return s.length; }));
                    // No contextual typing. Infers { (s: any): any }
```

The typing judgements for safeFTS have the pleasant property of unicity of typing; in other words, they define functions not relations.

**Lemma 2 (Unicity of typing).**

1. *If* $\Gamma \vdash \mathtt{e} : T_1$ *and* $\Gamma \vdash \mathtt{e} : T_2$ *then* $T_1 = T_2$.
2. *If* $\Gamma \vdash \mathtt{e} \downarrow S : T_1$ *and* $\Gamma \vdash \mathtt{e} \downarrow S : T_2$ *then* $T_1 = T_2$.

The proof of this lemma is by induction on typing derivations.

In safeFTS there are two typing relations for statements. We find it convenient to treat sequences of statements rather than single statements. The first typing relation, written $\Gamma \vdash \bar{\mathtt{s}} : \bar{\mathtt{R}}$, is read "given type environment $\Gamma$, the sequence of statements $\bar{\mathtt{s}}$ has (return) types $\bar{\mathtt{R}}$." The intention is that this judgement asserts both that the statements $\bar{\mathtt{s}}$ are well-typed and that the types $\bar{\mathtt{R}}$ are the types inferred for any `return` statements in the sequence (so the length of the type sequence $\bar{\mathtt{R}}$ is always less than or equal to the length of the statement sequence $\bar{\mathtt{s}}$). In line with the earlier discussion of scoping in JavaScript, it is assumed that when typing a sequence of statements $\bar{\mathtt{s}}$ the type environment contains types for all the identifiers declared in $\bar{\mathtt{s}}$.

**Statement sequence typing:** $\Gamma \vdash \bar{\mathtt{s}} : \bar{\mathtt{R}}$

$$[\text{I-EmpSeq}]\ \frac{}{\Gamma \vdash \bullet : \bullet} \qquad [\text{I-ExpSt}]\ \frac{\Gamma \vdash \mathtt{e} : \mathtt{S} \quad \Gamma \vdash \bar{\mathtt{s}} : \bar{\mathtt{R}}}{\Gamma \vdash \mathtt{e;}\ \bar{\mathtt{s}} : \bar{\mathtt{R}}}$$

$$[\text{I-If}]\ \frac{\Gamma \vdash \mathtt{e} : \mathtt{S} \quad \Gamma \vdash \bar{\mathtt{t}}_1 : \bar{\mathtt{T}}_1 \quad \Gamma \vdash \bar{\mathtt{t}}_2 : \bar{\mathtt{T}}_2 \quad \Gamma \vdash \bar{\mathtt{s}} : \bar{\mathtt{R}}}{\Gamma \vdash \mathtt{if\ (e)\ \{}\bar{\mathtt{t}}_1\mathtt{\}\ else\ \{}\bar{\mathtt{t}}_2\mathtt{\}}\ \bar{\mathtt{s}} : \bar{\mathtt{T}}_1, \bar{\mathtt{T}}_2, \bar{\mathtt{R}}}$$

$$[\text{I-Return}]\ \frac{\Gamma \vdash \bar{\mathtt{s}} : \bar{\mathtt{R}}}{\Gamma \vdash \mathtt{return;}\ \bar{\mathtt{s}} : \mathtt{void}, \bar{\mathtt{R}}} \qquad [\text{I-ReturnVal}]\ \frac{\Gamma \vdash \mathtt{e} : \mathtt{T} \quad \Gamma \vdash \bar{\mathtt{s}} : \bar{\mathtt{R}}}{\Gamma \vdash \mathtt{return\ e;}\ \bar{\mathtt{s}} : \mathtt{T}, \bar{\mathtt{R}}}$$

$$[\text{I-UTVarDec}]\ \frac{\Gamma(\mathtt{x}) = \mathtt{S} \quad \Gamma \vdash \bar{\mathtt{s}} : \bar{\mathtt{R}}}{\Gamma \vdash \mathtt{var\ x:S;}\ \bar{\mathtt{s}} : \bar{\mathtt{R}}}$$

$$[\text{I-ITVarDec}]\ \frac{\Gamma(\mathtt{x}) = \mathtt{S} \quad \Gamma \vdash \mathtt{e} \downarrow \mathtt{S} : \mathtt{T} \quad \mathtt{T} \leqq \mathtt{S} \quad \Gamma \vdash \bar{\mathtt{s}} : \bar{\mathtt{R}}}{\Gamma \vdash \mathtt{var\ x:S = e;}\ \bar{\mathtt{s}} : \bar{\mathtt{R}}}$$

$$[\text{I-UVarDec}]\ \frac{\Gamma(\mathtt{x}) = \mathtt{any} \quad \Gamma \vdash \bar{\mathtt{s}} : \bar{\mathtt{R}}}{\Gamma \vdash \mathtt{var\ x;}\ \bar{\mathtt{s}} : \bar{\mathtt{R}}}$$

$$[\text{I-IVarDec}]\ \frac{\mathtt{x} \in \mathit{dom}(\Gamma) \quad \Gamma \vdash \mathtt{e} : \mathtt{S} \quad \Gamma \uplus \mathtt{x} : \mathit{widen}(\mathtt{S}) \vdash \bar{\mathtt{s}} : \bar{\mathtt{R}}}{\Gamma \vdash \mathtt{var\ x = e;}\ \bar{\mathtt{s}} : \bar{\mathtt{R}}}$$

Rule [I-EmpSeq] asserts that the empty sequence is well typed. The rest of the rules are defined by the form of the first statement in the statement

sequence; they are routine, so we just describe the typing of return statements. In rule [I-Return] a `return` statement with no expression is well typed and has return type `void`. In rule [I-ReturnVal] a return statement `return e` is well typed and has the return type `T` if the expression `e` is of type `T`.

The second type relation for statement sequences is the analogue of contextual typing for expressions. It is written $\Gamma \vdash \bar{s} \downarrow T : \bar{R}$ and is read "given type environment $\Gamma$, the sequence of statements $\bar{s}$ in the context of type $T$ has (return) types $\bar{R}$." The intention is that this judgement captures both that the statements $\bar{s}$ are well typed and that the types $\bar{R}$ are the types inferred *in the context* of type $T$ for any return statements in the sequence.

**Statement sequence contextual typing:** $\Gamma \vdash \bar{s} \downarrow T : \bar{R}$

$$[\text{C-EmpSeq}]\ \frac{}{\Gamma \vdash \bullet \downarrow T : \bullet} \qquad [\text{C-ExpSt}]\ \frac{\Gamma \vdash e : S \quad \Gamma \vdash \bar{s} \downarrow T : \bar{R}}{\Gamma \vdash e;\ \bar{s} \downarrow T : \bar{R}}$$

$$[\text{C-If}]\ \frac{\Gamma \vdash e : S \quad \Gamma \vdash \bar{t}_1 \downarrow T : \bar{R}_1 \quad \Gamma \vdash \bar{t}_2 \downarrow T : \bar{R}_2 \quad \Gamma \vdash \bar{s} \downarrow T : \bar{R}_3}{\Gamma \vdash \texttt{if (e) } \{\bar{t}_1\} \texttt{ else } \{\bar{t}_2\}\ \bar{s} \downarrow T : \bar{R}_1, \bar{R}_2, \bar{R}_3}$$

$$[\text{C-Ret}]\ \frac{\Gamma \vdash \bar{s} \downarrow T : \bar{R}}{\Gamma \vdash \texttt{return};\ \bar{s} \downarrow T : \bar{R}} \qquad [\text{C-RetVal}]\ \frac{\Gamma \vdash e \downarrow T : S \quad S \leqq T \quad \Gamma \vdash \bar{s} \downarrow T : \bar{R}}{\Gamma \vdash \texttt{return } e;\ \bar{s} \downarrow T : S, \bar{R}}$$

$$[\text{C-UTVarDec}]\ \frac{\Gamma(x) = S \quad \Gamma \vdash \bar{s} \downarrow T : \bar{R}}{\Gamma \vdash \texttt{var } x{:}S;\ \bar{s} \downarrow T : \bar{R}}$$

$$[\text{C-ITVarDec}]\ \frac{\Gamma(x) = S \quad \Gamma \vdash e \downarrow S : S_1 \quad S_1 \leqq S \quad \Gamma \vdash \bar{s} \downarrow T : \bar{R}}{\Gamma \vdash \texttt{var } x{:}S = e;\ \bar{s} \downarrow T : \bar{R}}$$

$$[\text{C-UVarDec}]\ \frac{\Gamma(x) = \texttt{any} \quad \Gamma \vdash \bar{s} \downarrow T : \bar{R}}{\Gamma \vdash \texttt{var } x;\ \bar{s} \downarrow T : \bar{R}}$$

$$[\text{C-IVarDec}]\ \frac{\Gamma(x) = \texttt{any} \quad \Gamma \vdash e : S \quad \Gamma \uplus x : \mathit{widen}(S) \vdash \bar{s} \downarrow T : \bar{R}}{\Gamma \vdash \texttt{var } x = e;\ \bar{s} \downarrow T : \bar{R}}$$

Most of these rules are routine; the two important rules involve `return` statements. In rule [C-Ret] we capture the fact that JavaScript permits functions that return values to also contain return statements with no expressions. In rule [C-RetVal] a return statement `return e` is well typed and has return type `S` in the context of type `T` if the expression `e` in the context of type `T` has type `S` and that type `S` is assignable to type `T`.

## 5 Operational semantics

As explained in the introduction, the TypeScript compiler emits JavaScript code with no trace of the type system in the emitted code. So, the operational behaviour of TypeScript is just the behaviour of the underlying JavaScript implementation. However, in order to show that the safeFTS type system has the

desired safety properties we will give an operational semantics for TypeScript directly. We take as our starting point the operational semantics of Gardner et al. [7], although we make a number of simplifications.

A heap, $H$, is a partial function that maps a location $l$ to a heap object $o$. We assume a distinguished location `null`, which is not permitted to be in the domain of a heap. A heap object $o$ is either an object map (a partial function from variables to values, representing an object literal) or a closure. A variable $x$ is either a program variable `x`, a property name `n` or the internal property name `@this`. A value $v$ is either a location $l$ or a literal `l`. A closure is a pair consisting of a lambda expression (where we abbreviate `function (`$\bar{\texttt{x}}$`) { `$\bar{\texttt{s}}$` }` as $\lambda\bar{\texttt{x}}.\texttt{\{ }\bar{\texttt{s}}\texttt{ \}}$) and a scope chain $L$ (defined below).

We denote the empty heap by `emp`, a heap cell by $l \mapsto o$, the union of two disjoint heaps by $H_1 * H_2$, and a heap lookup by $H(l, x)$. We write heap update as $H[l \mapsto o]$, and where $o$ is an object map, we use the shorthand $H[(l, x) \mapsto v]$ to denote an update/extension to the $x$ element of the object map $o$.

JavaScript's dynamic semantics is complicated by the treatment of variables, which are not stored in an environment, but instead are resolved dynamically against an implicit scope object. A scope chain, $L$, is a list of locations of the scope objects, where we write $l : L$ for the list resulting from concatenating $l$ to the scope chain $L$. As safeFTS does not support `new` expressions, for simplicity, we do not model prototype lists. Function calls cause fresh local scope objects to be placed at the beginning of a scope chain and removed when the function body has been evaluated. All programs are evaluated with respect to a default scope chain $[l_g]$ where $l_g$ is the location of the global JavaScript object.

The lookup function $\sigma$ returns the location of the first scope object in the scope chain to define a given variable:

$$\sigma(H, l : L, x) \stackrel{\text{def}}{=} \begin{cases} l & \text{if } H(l, x) \downarrow \\ \sigma(H, L, x) & \text{otherwise} \end{cases}$$

A result $r$ can be either a value or a reference, which is a pair of a location and a variable; we make use of a function $\gamma$ where $\gamma(H, r)$ returns $r$ if $r$ is a value, and if it is a reference $(l, x)$ then it returns $H(l, x)$ if defined, or `undefined` if not.

The evaluation relation for FTS is written $\langle H_1, L, \texttt{e}\rangle \Downarrow \langle H_2, r\rangle$, which can be read "given initial heap $H_1$ and scope chain $L$, the expression `e` evaluates to a modified heap $H_2$ and a result $r$." We sometimes wish to dereference the result of evaluation, so we use the following shorthand $\langle H_1, L, \texttt{e}\rangle \Downarrow_{\text{v}} \langle H_2, v\rangle$ to mean that there exists a reference $r$ such that $\langle H_1, L, \texttt{e}\rangle \Downarrow \langle H_2, r\rangle$ and $\gamma(H_2, r) = v$.

**Expression evaluation:** $\langle H_1, L, \texttt{e}\rangle \Downarrow \langle H_2, r\rangle$

$$\text{[E-Id]}\ \frac{\sigma(H, L, \texttt{x}) = l}{\langle H, L, \texttt{x}\rangle \Downarrow \langle H, (l, \texttt{x})\rangle} \qquad \text{[E-Lit]}\ \frac{}{\langle H, L, \texttt{l}\rangle \Downarrow \langle H, \texttt{l}\rangle}$$

$$\text{[E-this]}\ \frac{\sigma(H, L, \texttt{@this}) = l_1 \qquad H(l_1, \texttt{@this}) = l}{\langle H, L, \texttt{this}\rangle \Downarrow \langle H, l\rangle}$$

$$\text{[E-ObLit]}\ \frac{\begin{array}{c} H_1 = H_0 * [l \mapsto \mathit{new}()] \\ \langle H_1, L, \mathtt{e}_1\rangle \Downarrow_{\mathrm{v}} \langle H_1', v_1\rangle \qquad H_2 = H_1'[(l, \mathtt{n}_1) \mapsto v_1] \\ \cdots \quad \langle H_m, L, \mathtt{e}_m\rangle \Downarrow_{\mathrm{v}} \langle H_m', v_m\rangle \qquad H = H_m'[(l, \mathtt{n}_m) \mapsto v_m] \end{array}}{\langle H_0, L, \{\ \mathtt{n}_1 \mathtt{:} \mathtt{e}_1 \mathtt{,}\ \ldots \mathtt{,} \mathtt{n}_m \mathtt{:} \mathtt{e}_m\ \}\rangle \Downarrow \langle H, l\rangle}$$

$$\text{[E-AssignExp]}\ \frac{\langle H_0, L, \mathtt{e}_1\rangle \Downarrow \langle H_1, (l, x)\rangle \qquad \langle H_1, L, \mathtt{e}_2\rangle \Downarrow_{\mathrm{v}} \langle H_2, v\rangle}{\langle H_0, L, \mathtt{e}_1 \mathtt{=} \mathtt{e}_2\rangle \Downarrow \langle H_2[(l, x) \mapsto v], v\rangle}$$

$$\text{[E-Op]}\ \frac{\langle H_0, L, \mathtt{e}_1\rangle \Downarrow_{\mathrm{v}} \langle H_1, \mathtt{l}_1\rangle \qquad \langle H_1, L, \mathtt{e}_2\rangle \Downarrow_{\mathrm{v}} \langle H_2, \mathtt{l}_2\rangle}{\langle H_0, L, \mathtt{e}_1 \oplus \mathtt{e}_2\rangle \Downarrow \langle H_2, \mathtt{l}_1 \oplus \mathtt{l}_2\rangle}$$

$$\text{[E-Prop]}\ \frac{\langle H_0, L, \mathtt{e}\rangle \Downarrow_{\mathrm{v}} \langle H_1, l\rangle \qquad l \neq \texttt{null}}{\langle H_0, L, \mathtt{e.n}\rangle \Downarrow \langle H_1, (l, \mathtt{n})\rangle}$$

$$\text{[E-Prop']}\ \frac{\langle H_0, L, \mathtt{e}\rangle \Downarrow_{\mathrm{v}} \langle H_1, \mathtt{l}\rangle \qquad H_2 = H_1 * [l \mapsto \mathit{box}(\mathtt{l})]}{\langle H_0, L, \mathtt{e.n}\rangle \Downarrow \langle H_2, (l, \mathtt{n})\rangle}$$

$$\text{[E-CompProp]}\ \frac{\langle H_0, L, \mathtt{e}\rangle \Downarrow_{\mathrm{v}} \langle H_1, l\rangle \qquad l \neq \texttt{null} \qquad \langle H_1, L, \mathtt{f}\rangle \Downarrow_{\mathrm{v}} \langle H_2, \ulcorner \mathtt{n} \urcorner\rangle}{\langle H_0, L, \mathtt{e[f]}\rangle \Downarrow \langle H_2, (l, \mathtt{n})\rangle}$$

$$\text{[E-CompProp']}\ \frac{\begin{array}{c} \langle H_0, L, \mathtt{e}\rangle \Downarrow_{\mathrm{v}} \langle H_1, \mathtt{l}\rangle \qquad H_2 = H_1 * [l \mapsto \mathit{box}(\mathtt{l})] \\ \langle H_2, L, \mathtt{f}\rangle \Downarrow_{\mathrm{v}} \langle H_3, \ulcorner \mathtt{n} \urcorner\rangle \end{array}}{\langle H_0, L, \mathtt{e[f]}\rangle \Downarrow \langle H_3, (l, \mathtt{n})\rangle}$$

$$\text{[E-Call]}\ \frac{\begin{array}{ccc} \langle H_0, L_0, \mathtt{e}\rangle \Downarrow \langle H_1, r\rangle & & \gamma(H_1, r) = l_1 \\ H(l_1) = \langle \lambda \bar{\mathtt{x}}.\{\bar{\mathtt{s}}\}, L_1\rangle & & \mathsf{This}(H_1, r) = l_2 \\ \langle H_1, L_0, \mathtt{e}_1\rangle \Downarrow_{\mathrm{v}} \langle H_2, v_1\rangle & \cdots & \langle H_n, L_0, \mathtt{e}_n\rangle \Downarrow_{\mathrm{v}} \langle H_{n+1}, v_n\rangle \\ H' = H_{n+1} * \mathit{act}(l, \bar{\mathtt{x}}, \bar{v}, \{\bar{\mathtt{s}}\}, l_2) & & \langle H', l : L_1, \bar{\mathtt{s}}\rangle \Downarrow \langle H'', \texttt{return}\ v\mathtt{;}\rangle \end{array}}{\langle H_0, L_0, \mathtt{e(e}_1\mathtt{,}\ \ldots\mathtt{,}\mathtt{e}_n\mathtt{)}\rangle \Downarrow \langle H'', v\rangle}$$

$$\text{[E-CallUndef]}\ \frac{\begin{array}{ccc} \langle H_0, L_0, \mathtt{e}\rangle \Downarrow \langle H_1, r\rangle & & \gamma(H_1, r) = l_1 \\ H(l_1) = \langle \lambda \bar{\mathtt{x}}.\{\bar{\mathtt{s}}\}, L_1\rangle & & \mathsf{This}(H_1, r) = l_2 \\ \langle H_1, L_0, \mathtt{e}_1\rangle \Downarrow_{\mathrm{v}} \langle H_2, v_1\rangle & \cdots & \langle H_n, L_0, \mathtt{e}_n\rangle \Downarrow_{\mathrm{v}} \langle H_{n+1}, v_n\rangle \\ H' = H_{n+1} * \mathit{act}(l, \bar{\mathtt{x}}, \bar{v}, \{\bar{\mathtt{s}}\}, l_2) & & \langle H', l : L_1, \bar{\mathtt{s}}\rangle \Downarrow \langle H'', \texttt{return;}\rangle \end{array}}{\langle H_0, L_0, \mathtt{e(e}_1\mathtt{,}\ \ldots\mathtt{,}\mathtt{e}_n\mathtt{)}\rangle \Downarrow \langle H'', \texttt{undefined}\rangle}$$

$$\text{[E-Func]}\ \frac{H_1 = H_0 * [l \mapsto \langle \lambda \bar{\mathtt{x}}.\{\ \bar{\mathtt{s}}\ \}, L\rangle]}{\langle H_0, L, \texttt{function (}\bar{\mathtt{x}}\texttt{) \{ }\bar{\mathtt{s}}\texttt{ \}}\rangle \Downarrow \langle H_1, l\rangle}$$

$$\text{[E-TypeAssert]}\ \frac{\langle H_0, L, \mathtt{e}\rangle \Downarrow \langle H_1, r_1\rangle}{\langle H_0, L, \texttt{<T>e}\rangle \Downarrow \langle H_1, r_1\rangle}$$

Most of these rules are routine; Gardner et al. [7] give extensive details. We restrict our attention to just a few of the more important rules. In rule [E-ObLit] we create at fresh location $l$ a new object map (using an auxiliary function *new*) and update its elements in order. In rule [E-CompProp] we require that the

property subexpression $\mathtt{f}$ evaluates to a string that denotes a name $\mathtt{n}$; this string we write as $\ulcorner \mathtt{n} \urcorner$. In rules [E-Prop'] and [E-CompProp'] we cover the case where properties are accessed on primitive values (which are implicitly boxed, using an auxiliary function *box*). In rule [E-Call] the important step is that we create a fresh local scope object (stored at location $l'$) with which we evaluate the body of the function. We make use of an auxiliary function This (taken from [7, §3.3]) that captures the behaviour of the `this` keyword, and an auxiliary function *act* that builds the new scope object.

$$\mathsf{This}(H, (l, x)) \stackrel{\text{def}}{=} l \text{ if } H(l, @\mathtt{this}) \downarrow; \text{ and } \mathsf{This}(H, v) \stackrel{\text{def}}{=} l_g \text{ otherwise}$$
$$act(l, \bar{\mathtt{x}}, \bar{v}, \bar{\mathtt{s}}, l') \stackrel{\text{def}}{=} l \mapsto (\{\bar{\mathtt{x}} \mapsto \bar{v}, @\mathtt{this} \mapsto l'\} * defs(\bar{\mathtt{x}}, l, \bar{\mathtt{s}}))$$

The auxiliary function *defs* searches the statements $\bar{\mathtt{s}}$ for all the declared variables and makes them in scope in the current local scope object; this is the operational counterpart to the "awful" feature of JavaScript scoping described in §4. Rule [E-CallUndef] reflects the JavaScript semantics that functions that do not specify a return value actually return the `undefined` value.

The evaluation relation for statement sequences is of the form $\langle H_0, L, \bar{\mathtt{s}}_0 \rangle \Downarrow \langle H_1, s \rangle$ where $s$ is a statement result, which is a statement of the form `return;`, `return` $v$`;`, or `;`. The rules for evaluating statements are routine and omitted.

In order to prove type soundness, we need to extend the notion of typing to the operational semantics (in the style of [1, 3]). A heap type $\Sigma$ is a partial function from locations to types (which are either function types or object literal types). The statement of subject reduction then relies on a number of new judgements. First, we need a well-formedness relation for a heap $H$, written $H \models \diamond$. We also need a judgement that a heap $H$ and scope chain $L$ are compatible, written $H, L \models \diamond$, which essentially means that all the scope objects in the scope chain exist in the heap. We use a judgement written $\Sigma \models H$ that captures that a heap $H$ is compatible with a heap type $\Sigma$. We also make use of a function $context(\Sigma, L)$ that builds a typing judgement corresponding to the variables in the scope chain $L$, using their types stored in $\Sigma$. Using these judgements, we can then write $\Sigma \models \langle H, L, \mathtt{e} \rangle : \mathtt{T}$ to mean $\Sigma \models H$, $H, L \models \diamond$ and $context(\Sigma, L) \vdash \mathtt{e} : \mathtt{T}$. Similarly we can define judgements $\Sigma \models \langle H, L, \mathtt{e} \rangle \downarrow \mathtt{S} : \mathtt{T}$, $\Sigma \models \langle H, L, \bar{\mathtt{s}} \rangle : \bar{\mathtt{T}}$ and $\Sigma \models \langle H, L, \bar{\mathtt{s}} \rangle \downarrow \mathtt{S} : \bar{\mathtt{T}}$. Finally, we can define two judgements on results of evaluation, written $\Sigma \models \langle H, r \rangle : \mathtt{T}$ and $\Sigma \models \langle H, r \rangle \downarrow \mathtt{S} : \mathtt{T}$ (along with variants for statement results). We write $\Sigma \subseteq \Sigma'$ to mean that $\Sigma'$ is an extension of $\Sigma$ in the usual sense.

**Theorem 1 (Subject reduction).**

1. *If* $\Sigma \models \langle H, L, \mathtt{e} \rangle : T$ *and* $\langle H, L, \mathtt{e} \rangle \Downarrow \langle H', r \rangle$ *then* $\exists \Sigma', T'$ *such that* $\Sigma \subseteq \Sigma'$, $\Sigma' \models \langle H', r \rangle : T'$ *and* $T' \leqq T$.
2. *If* $\Sigma \models \langle H, L, \mathtt{e} \rangle \downarrow S : T$ *and* $\langle H, L, \mathtt{e} \rangle \Downarrow \langle H', r \rangle$ *then* $\exists \Sigma', T'$ *such that* $\Sigma \subseteq \Sigma'$, $\Sigma' \models \langle H', r \rangle \downarrow S : T'$ *and* $T' \leqq T$.
3. *If* $\Sigma \models \langle H, L, \bar{\mathtt{s}} \rangle : \bar{T}$ *and* $\langle H, L, \bar{\mathtt{s}} \rangle \Downarrow \langle H', s \rangle$ *then* $\exists \Sigma', T'$ *such that* $\Sigma \subseteq \Sigma'$, $\Sigma' \models \langle H', s \rangle : T'$ *and* $T' \leqq return(\bar{T})$.
4. *If* $\Sigma \models \langle H, L, \bar{\mathtt{s}} \rangle \downarrow S : \bar{T}$ *and* $\langle H, L, \bar{\mathtt{s}} \rangle \Downarrow \langle H', s \rangle$ *then* $\exists \Sigma', T'$ *such that* $\Sigma \subseteq \Sigma'$, $\Sigma' \models \langle H', s \rangle \downarrow S : T'$ *and* $T' \leqq return(\bar{T})$.

## 6 Production Featherweight TypeScript (prodFTS)

In this section we define prodFTS which can be viewed as the core calculus of the full TypeScript language. We define it as a series of extensions to the type system of safeFTS. Each of these extensions is unsound. We organize them according to the source of unsoundness, along the lines suggested in §2.

### 6.1 Unchecked downcasts

In addition to the upcasts allowed in safeFTS, prodFTS also supports downcasts.

$$\frac{\Gamma \vdash \texttt{e}: \texttt{S} \quad \texttt{T} \leqq \texttt{S}}{\Gamma \vdash \texttt{<T>e}: \texttt{T}}$$

Unlike in languages such as Java and C♯, these downcasts are not automatically checked at runtime, because all type information is erased by the compiler. The following example illustrates this issue:

```
interface Shape { ... }
interface Circle extends Shape { ... }
interface Triangle extends Shape { ... }
function createShape(kind: string): Shape {
    if (kind === "circle") return buildCircle();
    if (kind === "triangle") return buildTriangle();
    ... }
var circle = <Circle> createShape("circle");
```

Here, the TypeScript type system will rely on the fact that, after the type assertion, `circle` is of type `Circle`. The responsibility of guarding against erroneous creation of, for example a `Triangle`, remains with the programmer. Should runtime checks be needed, the TypeScript programmer would have to simulate them using JavaScript's introspection capabilities.

### 6.2 Unchecked gradual typing (and unchecked indexing)

TypeScript has a gradual type system in the style of Siek and Taha [14]. However, unlike most languages with gradual type systems, dynamic checks are not made to ensure safety (again, because types are removed by the compiler).

The key to gradual type systems is that the `any` type is treated specially. This type serves as the boundary between the statically typed world (code typed without reference to `any`) and the dynamically typed world. The fundamental feature of `any` is that any type can be implicitly converted to `any` and `any` can be implicitly converted to any other type. The former of these conversions is allowed in safeFTS via the rule [A-AnyR]. prodFTS includes the following additional rule in order to support conversions in the opposite direction:

$$\frac{\texttt{T} \vdash \diamond}{\texttt{any} \leqq \texttt{T}}$$

This extension to assignment compatibility is quite drastic. In particular, assignment compatibility is no longer transitive! For example, we now have that `string` $\leqq$ `any` and `any` $\leqq$ `boolean` but not that `string` $\leqq$ `boolean`. Moreover, this extension implies that assignment compatibility is not a good basis for determining best common types or for overloading resolution. Therefore, TypeScript introduces a new type relation, called subtyping. In contrast to the definition of assignment compatibility, it is not the case that `any` is a subtype of any other type. In all other respects, however, subtyping is defined identically to assignment compatibility [11, §3.8.2]. Accordingly, in prodFTS, subtyping replaces assignment compatibility in the definitions of best common types and for overloading resolution.

Furthermore, TypeScript allows the liberal use of subexpressions of type `any`. (Such use is how gradual type systems permit the mixing of dynamic and statically-typed code.) In particular, those subexpressions may be used for potentially unsafe indexing. We capture this aspect of TypeScript by including the following extra typing rules in prodFTS:

$$\frac{\Gamma \vdash \texttt{e} : \texttt{any}}{\Gamma \vdash \texttt{e.n} : \texttt{any}} \qquad \frac{\Gamma \vdash \texttt{e} : \texttt{any} \quad \Gamma \vdash \bar{\texttt{f}} : \bar{\texttt{S}}}{\Gamma \vdash \texttt{e}(\bar{\texttt{f}}) : \texttt{any}}$$

$$\frac{\Gamma \vdash \texttt{e} : \texttt{any} \quad \Gamma \vdash \texttt{f} : \texttt{string}}{\Gamma \vdash \texttt{e[f]} : \texttt{any}} \qquad \frac{\Gamma \vdash \texttt{e} : \texttt{T} \quad \Gamma \vdash \texttt{f} : \texttt{any}}{\Gamma \vdash \texttt{e[f]} : \texttt{any}}$$

Siek and Taha employ occurrences of these rules in order to inject runtime checks into code, with the goal of ensuring that the code satisfies type contracts. Once more, as TypeScript removes all type information, analogous checks are not made in TypeScript, so runtime type errors are possible.

### 6.3 Unchecked covariance

As mentioned in the introduction, TypeScript was designed as a language to which existing JavaScript programmers could migrate in a seamless way. In particular, existing libraries and codebases can be given type signatures without disturbing the source code. (An alternative approach would be to require programmers to restructure their code so particular features of some new type system could be used to greater effect.) Therefore, common programming idioms must be supported directly at the type level. One such idiom that occurs extensively in JavaScript codebases and thus is supported directly is covariance of property and parameter types in function signatures. Although this idiom is not in general safe, dynamic programmers frequently make safe use of it. For instance (much as in [1]), consider a program that uses the types `Person` and `Vegetarian`. In `Person`, a member `eat` takes arguments of type `any`; in `Vegetarian`, it takes arguments of a type `Vegetables`, which is also the type of another member `myLunch`. Covariance allows `Vegetarian` to be assignable to `Person`, and errors won't arise as long as objects of type `Vegetarian` are fed the contents of `myLunch`.

In prodFTS, we capture covariance via a revised notion of assignment compatibility of members, with the following rules:

$$\frac{\mathtt{S} \leqq \mathtt{T}}{\mathtt{n{:}S} \leqq \mathtt{n{:}T}} \qquad \frac{\bar{\mathtt{S}} \cong \bar{\mathtt{T}} \qquad \mathtt{R}_1 \neq \mathtt{void} \qquad \mathtt{R}_0 \leqq \mathtt{R}_1}{(\bar{\mathtt{x}}{:}\bar{\mathtt{S}}){:}\mathtt{R}_0 \leqq (\bar{\mathtt{y}}{:}\bar{\mathtt{T}}){:}\mathtt{R}_1} \qquad \frac{\bar{\mathtt{S}} \cong \bar{\mathtt{T}} \qquad \mathtt{R} \vdash \diamond}{(\bar{\mathtt{x}}{:}\bar{\mathtt{S}}){:}\mathtt{R} \leqq (\bar{\mathtt{y}}{:}\bar{\mathtt{T}}){:}\mathtt{void}}$$

The first rule permits covariance on member typing. The others permit call signatures to be bivariant (either covariant or contravariant) in their argument types and covariant in their result types (where $\mathtt{S} \cong \mathtt{T} \stackrel{\text{def}}{=} \mathtt{S} \leqq \mathtt{T}$ or $\mathtt{T} \leqq \mathtt{S}$).

## 7 Connection to gradual typing

In this section, we aim to give precise substance to our claim that TypeScript is a gradual type system in the style of Siek and Taha [14]. Specifically, we define a notion of consistent-subtyping for TypeScript types, and prove that it is equivalent to the notion of assignment compatibility in prodFTS, defined in the previous section.

Our first step is to define a restriction operator on types and members. Basically, $\mathtt{S}|_{\mathtt{T}}$ masks off the parts of $\mathtt{S}$ that are unknown (i.e., any) in $\mathtt{T}$.

$$\mathtt{S}|_{\mathtt{T}} \stackrel{\text{def}}{=} \begin{cases} \mathtt{any} & \text{if } \mathtt{T} \equiv \mathtt{any} \\ \{\ \bar{\mathtt{M}}_0|_{\bar{\mathtt{M}}_1}\ \} & \text{if } \mathtt{S} \equiv \{\ \bar{\mathtt{M}}_0\ \} \text{ and } \mathtt{T} \equiv \{\ \bar{\mathtt{M}}_1, \bar{\mathtt{M}}_2\ \} \\ \{\ \bar{\mathtt{M}}_0|_{\bar{\mathtt{M}}_2}, \bar{\mathtt{M}}_1\ \} & \text{if } \mathtt{S} \equiv \{\ \bar{\mathtt{M}}_0, \bar{\mathtt{M}}_1\ \} \text{ and } \mathtt{T} \equiv \{\ \bar{\mathtt{M}}_2\ \} \\ \mathtt{S} & \text{otherwise} \end{cases}$$

$$\mathtt{M}_0|_{\mathtt{M}_1} \stackrel{\text{def}}{=} \begin{cases} \mathtt{n}\colon \mathtt{S}|_{\mathtt{T}} & \text{if } \mathtt{M}_0 \equiv \mathtt{n{:}S} \text{ and } \mathtt{M}_1 \equiv \mathtt{n{:}T} \\ (\bar{\mathtt{x}}{:}\bar{\mathtt{S}}_0|_{\bar{\mathtt{S}}_1}){:}\mathtt{T}_0|_{\mathtt{T}_1} & \text{if } \mathtt{M}_0 \equiv (\bar{\mathtt{x}}{:}\bar{\mathtt{S}}_0){:}\mathtt{T}_0 \text{ and } \mathtt{M}_1 \equiv (\bar{\mathtt{x}}{:}\bar{\mathtt{S}}_1){:}\mathtt{T}_1 \end{cases}$$

Next we introduce a simple subtyping relation. This relation, written $\mathtt{S} <: \mathtt{T}$ (and $\mathtt{M}_0 <: \mathtt{M}_1$ on members), gives no special status to the type any. It is covariant in members and, for call signatures, bivariant in argument types and covariant in return types. (We write $\mathtt{S} <:> \mathtt{T}$ to mean either $\mathtt{S} <: \mathtt{T}$ or $\mathtt{T} <: \mathtt{S}$.)

$$\frac{\mathtt{S} \vdash \diamond}{\mathtt{S} <: \mathtt{S}} \qquad \frac{\mathtt{T} \vdash \diamond}{\mathit{Undefined} <: \mathtt{T}} \qquad \frac{\mathtt{T} \vdash \diamond \qquad \mathtt{T} \neq \mathit{Undefined}}{\mathit{Null} <: \mathtt{T}}$$

$$\frac{\mathcal{I}(\mathtt{P}) <: \mathtt{T}}{\mathtt{P} <: \mathtt{T}} \qquad \frac{\{\ \bar{\mathtt{M}}_0, \bar{\mathtt{M}}_1\ \} \vdash \diamond \qquad \bar{\mathtt{M}}_1 <: \bar{\mathtt{M}}_2}{\{\ \bar{\mathtt{M}}_0, \bar{\mathtt{M}}_1\ \} <: \{\ \bar{\mathtt{M}}_2\ \}} \qquad \frac{\mathtt{S} <: \mathtt{T}}{\mathtt{n{:}S} <: \mathtt{n{:}T}}$$

$$\frac{\bar{\mathtt{S}} <:> \bar{\mathtt{T}} \qquad \mathtt{R}_1 \neq \mathtt{void} \qquad \mathtt{R}_0 <: \mathtt{R}_1}{(\bar{\mathtt{x}}{:}\bar{\mathtt{S}}){:}\mathtt{R}_0 <: (\bar{\mathtt{y}}{:}\bar{\mathtt{T}}){:}\mathtt{R}_1} \qquad \frac{\bar{\mathtt{S}} <:> \bar{\mathtt{T}} \qquad \mathtt{R} \vdash \diamond}{(\bar{\mathtt{x}}{:}\bar{\mathtt{S}}){:}\mathtt{R} <: (\bar{\mathtt{y}}{:}\bar{\mathtt{T}}){:}\mathtt{void}}$$

Then, following Siek and Taha, we define consistent-subtyping, written $\mathtt{S} \lesssim \mathtt{T}$, as $\mathtt{S}|_{\mathtt{T}} <: \mathtt{T}|_{\mathtt{S}}$. The following theorem expresses the equivalence of consistent-subtyping and assignment compatibility. Its proof, which we omit, relies on coinduction.

**Theorem 2.** *$S \lesssim T$ if and only if $S \leqq T$.*

## 8 Related work

Since JavaScript's recent rise to prominence, there has been considerable work on providing a suitable type system for the language. Here we can only mention a subset of that work. Various research efforts have explored sound approaches to this problem. Thiemann [17] proposed an early type system that uses singleton types and first-class record labels, and in the same year Anderson et al. [2] proposed another type system with a focus on type inference. A number of others have proposed systems of increasing complexity to deal with the complicated programming patterns found in JavaScript; for example, Chugh et al. [4] employed nested refinements and heap types in DJS, and Guha et al. [9] proposed a combination of a type system and a flow analysis.

Others have emphasized support for development at scale. In particular, like TypeScript, the Dart language [8] relaxes soundness in order to support dynamic programming. Dart is closely related to JavaScript, and can also compile directly to JavaScript in such a way that all traces of the type system are removed. However, unlike TypeScript, Dart is an entirely new language.

Whilst TypeScript favours convenience over soundness, our work can be used as the basis for defining safe variants of TypeScript. Bhargavan et al. [16] extend a similar safe fragment with a new type to denote values from untrusted JavaScript code and employ runtime type information instead of type erasure, focusing on using type-driven wrappers to ensure important security properties.

Further afield, various dynamic languages have been extended with type systems. For instance, Typed Scheme [18] adds a type system to Scheme. It introduces a notion of occurrence typing and combines a number of type system features such as recursive types, union types, and polymorphism.

## 9 Conclusion

This paper describes and analyses the core of the TypeScript language, and in particular its type system. The work that it represents has been useful in resolving ambiguities in the language definition, and in identifying minor unintended inconsistencies and mistakes in the language implementation. It provides a basis for partial soundness theorems, and it isolates and accounts for sources of unsoundness in the type system.

Beyond the details of this work (which are specific to TypeScript, and which may perhaps change, as TypeScript develops further), we hope that our results will contribute to the principled study of deliberate unsoundness. In this direction, we believe that there are various opportunities for intriguing further research. In particular, to the extent that any type system expresses programmer intent, we would expect that it could be useful in debugging, despite its unsoundness. Research on blame, e.g., [19], might be helpful in this respect. It may also be worthwhile to codify programmer guidance that would, over time, reduce the reliance on dangerous typing rules. Static analysis tools may support this guidance and complement a type system. These and related projects would

aim to look beyond sound language fragments: the principles of programming languages may also help us understand and live with unsoundness.

## References


1. M. Abadi and L. Cardelli. *A theory of objects*. Springer Verlag, 1996.
2. C. Anderson, P. Giannini, and S. Drossopoulou. Towards type inference for JavaScript. In *Proceedings of ECOOP*, 2005.
3. G. Bierman, M. Parkinson, and A. Pitts. MJ: An imperative core calculus for Java and Java with effects. Technical Report 563, University of Cambridge Computer Laboratory, 2003.
4. R. Chugh, D. Herman, and R. Jhala. Dependent types for JavaScript. In *Proceedings of OOSLA*, 2012.
5. D. Crockford. *JavaScript: The good parts*. O'Reilly, 2008.
6. V. Gapeyev, M. Levin, and B. Pierce. Recursive subtyping revealed. *JFP*, 12(6):511–548, 2002.
7. P. Gardner, S. Maffeis, and G. Smith. Towards a program logic for JavaScript. In *Proceedings of POPL*, 2013.
8. Google. Dart programming language. `http://www.dartlang.org`.
9. A. Guha, C. Saftoiu, and S. Krisnamurthi. Typing local control and state using flow analysis. In *Proceedings of ESOP*, 2011.
10. A. Igarashi, B. Pierce, and P. Wadler. Featherweight Java: A minimal core calculus for Java and GJ. *ACM TOPLAS*, 23(3):396–450, 2001.
11. Microsoft Corporation. *TypeScript Language Specification*, 0.9.5 edition, 2014. `http://typescriptlang.org`.
12. B. Pierce and D. Turner. Local type inference. In *Proceedings of POPL*, 1998.
13. G. Sénizergues. The equivalence problem for deterministic pushdown automata is decidable. In *Proceedings of ICALP*, 1997.
14. J. Siek and W. Taha. Gradual typing for objects. In *Proceedings of ECOOP*, 2007.
15. M. H. Solomon. Type definitions with parameters. In *Proceedings of POPL*, 1978.
16. N. Swamy, C. Fournet, A. Rastogi, K. Bhargavan, J. Chen, P.-Y. Strub, and G. Bierman. Gradual typing embedded securely in JavaScript. In *Proceedings of POPL*, 2014.
17. P. Thiemann. Towards a type systems for analyzing JavaScript programs. In *Proceedings of ESOP*, 2005.
18. S. Tobin-Hochstadt and M. Felleisen. The design and implementation of Typed Scheme. In *Proceedings of POPL*, 2008.
19. P. Wadler and R. B. Findler. Well-typed programs can't be blamed. In *Proceedings of ESOP*, 2009.