Safe & Efficient Gradual Typing for TypeScript

Aseem Rastogi * — University of Maryland College Park — aseem@cs.umd.edu

Nikhil Swamy, Cédric Fournet — Microsoft Research — {nswamy, fournet}@microsoft.com

Gavin Bierman * — Oracle Labs — Gavin.Bierman@oracle.com

Panagiotis Vekris * — University of California San Diego — pvekris@cs.ucsd.edu

Abstract

Current proposals for adding gradual typing to JavaScript, such as Closure, TypeScript and Dart, forgo soundness to deal with issues of scale, code reuse, and popular programming patterns.

We show how to address these issues in practice while retaining soundness. We design and implement a new gradual type system, prototyped for expediency as a 'Safe' compilation mode for TypeScript. Our compiler achieves soundness by enforcing stricter static checks and embedding residual runtime checks in compiled code. It emits plain JavaScript that runs on stock virtual machines. Our main theorem is a simulation that ensures that the checks introduced by Safe TypeScript (1) catch any dynamic type error, and (2) do not alter the semantics of type-safe TypeScript code.

Safe TypeScript is carefully designed to minimize the performance overhead of runtime checks. At its core, we rely on two new ideas: *differential subtyping*, a new form of coercive subtyping that computes the minimum amount of runtime type information that must be added to each object; and an *erasure modality*, which we use to safely and selectively erase type information. This allows us to scale our design to full-fledged TypeScript, including arrays, maps, classes, inheritance, overloading, and generic types.

We validate the usability and performance of Safe TypeScript empirically by type-checking and compiling around 120,000 lines of existing TypeScript source code. Although runtime checks can be expensive, the end-to-end overhead is small for code bases that already have type annotations. For instance, we bootstrap the Safe TypeScript compiler (90,000 lines including the base TypeScript compiler): we measure a 15% runtime overhead for type safety, and also uncover programming errors as type safety violations. We conclude that, at least during development and testing, subjecting JavaScript/TypeScript programs to safe gradual typing adds significant value to source type annotations at a modest cost.

1. Introduction

Originally intended for casual scripting, JavaScript is now widely used to develop large applications. Using JavaScript in complex codebases is, however, not without difficulties: the lack of robust language abstractions such as static types, classes, and interfaces can hamper programmer productivity and undermine tool support. Unfortunately, retrofitting abstraction into JavaScript is difficult, as one must support awkward language features and programming patterns in existing code and third-party libraries, without either rejecting most programs or requiring extensive annotations (perhaps using a PhD-level type system).

Gradual type systems set out to fix this problem in a principled manner, and have led to popular proposals for JavaScript, notably Closure, TypeScript and Dart (although the latter is strictly speaking not JavaScript but a variant with some features of JavaScript removed). These proposals bring substantial benefits to the working programmer, usually taken for granted in typed languages, such as a convenient notation for documenting code; API exploration; code completion; refactoring; and diagnostics of basic type errors. Interestingly, to be usable at scale, all these proposals are *intentionally unsound*: typeful programs may be easier to write and maintain, but their type annotations do not prevent runtime type errors.

Instead of giving up on soundness at the outset, we contend that a sound gradual type system for JavaScript is practically feasible. There are, undoubtedly, some significant challenges to overcome. For starters, the language includes inherently type-unsafe features such as eval and stack walks, some of JavaScript's infamous "bad parts". However, recent work is encouraging: Swamy et al. (2014) proposed TS* a sound type system for JavaScript to tame untyped adversarial code, isolating it from a gradually typed core language. Although the typed fragment of TS* is too limited for large-scale JavaScript developments, its recipe of coping with the bad parts using type-based memory isolation is promising.

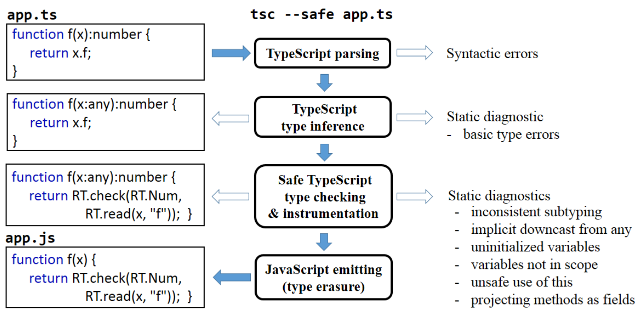
In this work, we tackle the problem of developing a sound, yet practical, gradual type system for a large fragment of JavaScript, confining its most awkward features to untrusted code by relying implicitly on memory isolation. Concretely, we take TypeScript as our starting point. In brief, TypeScript is JavaScript with optional type annotations: every valid JavaScript program is a valid TypeScript program. TypeScript adds an object-oriented gradual type system, while its compiler erases all traces of types and emits JavaScript that can run on stock virtual machines. The emitted code is syntactically close to the source (except for type erasure), hence TypeScript and JavaScript interoperate with the same performance.

TypeScript's type system is intentionally unsound; Bierman et al. (2014) catalog some of its unsound features, including bivariant subtyping for functions and arrays, as well as in class and interface extension. The lack of soundness limits the benefits of writing type annotations in TypeScript, making abstractions hard to enforce and leading to unconventional programming patterns, even for programmers who steer clear of the bad parts. Consider for instance the following snippet from TouchDevelop (Tillmann et al. 2012), a mobile programming platform written in TypeScript:

```
private parseColorCode (c:string) {
  if (typeof c !== "string") return −1; . . . }
```

* This work was done at Microsoft Research.

POPL '15, January 15–17, 2015, Mumbai, India.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-3300-9/15/01. . . $15.00.
http://dx.doi.org/10.1145/2676726.2676971



```
app.ts
function f(x):number {
    return x.f;
}
```

```
function f(x:any):number {
    return x.f;
}
```

```
function f(x:any):number {
    return RT.check(RT.Num,
                RT.read(x, "f")); }
```

```
app.js
function f(x) {
    return RT.check(RT.Num,
                RT.read(x, "f")); }
```

tsc --safe app.ts
TypeScript parsing
Syntactic errors
TypeScript type inference
Static diagnostic
- basic type errors
Safe TypeScript type checking & instrumentation
Static diagnostics
- inconsistent subtyping
- implicit downcast from any
- uninitialized variables
- variables not in scope
- unsafe use of this
- projecting methods as fields
JavaScript emitting (type erasure)

Figure 1: Architecture of Safe TypeScript

Despite annotating the formal parameter c as a string, the prudent TypeScript programmer must still check that the argument received is indeed a string using JavaScript reflection, and deal with errors.

1.1 Safe TypeScript

We present a new type-checker and code generator for a subset of TypeScript that guarantees type safety through a combination of static and dynamic checks. Its implementation is fully integrated as a branch of the TypeScript-0.9.5 compiler. Programmers can opt in to Safe TypeScript simply by providing a flag to the compiler (similar in spirit to JavaScript's strict mode, which lets the programmer abjure some unsafe features). As with TypeScript, the code generated by Safe TypeScript is standard JavaScript and runs on stock virtual machines.

Figure 1 illustrates Safe TypeScript at work. A programmer authors a TypeScript program, `app.ts`, and feeds it to the TypeScript compiler, `tsc`, setting the `--safe` flag to enable our system. The compiler initially processes `app.ts` using standard TypeScript passes: the file is parsed and a type inference algorithm computes (potentially unsound) types for all subterms. For the top-level function f in the figure, TypeScript infers the type (x:any)⇒number, using by default the dynamic type any for its formal parameter. (It may infer more precise types in other cases.) The sub-term x.f is inferred to have type any as well. In TypeScript, any-typed values can be passed to a context expecting a more precise type, so TypeScript silently accepts that x.f be returned at type number. Since TypeScript erases all types, x.f need not be a number at runtime, which may cause callers of f to fail later, despite f's annotation.

In contrast, when using Safe TypeScript, a second phase of type-checking is applied to the program, to confirm (soundly) the types inferred by earlier phases. This second phase may produce various static errors and warnings. Once all static errors have been fixed, Safe TypeScript rewrites the program to instrument objects with runtime type information (RTTI) and insert runtime checks based on this RTTI. In the example, the rewriting involves instrumenting x.f as RT.readField(x, "f"), a call into a runtime library RT used by all Safe TypeScript programs. Although the static type of x is any, the RTTI introduced by our compiler allows the runtime library to determine whether it is safe to project x.f, and further (using RT.check) to ensure that its content is indeed a number. Finally, the dynamically type-safe JavaScript code is emitted by a code generator that strips out type annotations and desugars constructs like classes, but otherwise leaves the program unchanged.

Underlying Safe TypeScript are two novel technical ideas:

(1) Partial erasure. Many prior gradual type systems require that a single dynamic type (variously called dynamic, dyn, $*$, any, $\top$, etc.) be a universal super-type and, further, that any be related to all other types by subtyping and coercion. We relax this requirement: in Safe TypeScript, any characterizes only those values that are tagged with RTTI. Separately, we have a modality for *erased types*, whose values need not be tagged with RTTI. Erased types are not subtypes of any, nor can they be coerced to it, yielding four important capabilities. First, *information hiding*: we show how to use erased types to encode private fields in an object and prove a confidentiality theorem (Theorem 2). Second, *user-controlled performance*: through careful erasure, the user can minimize the overhead of RTTI operations. Third, *modularity*: erased-types allow us to ensure that objects owned by external modules do not have RTTI. And, fourth, *long-term evolution*, allowing us to scale Safe TypeScript up to a language with a wide range of typing features.

(2) Differential subtyping. In addition, we rely on a form of coercive subtyping (Luo 1999) that allows us to attach *partial* RTTI on any-typed objects, and is vital for good runtime performance.

1.2 Main contributions

We present the first sound gradual type system with a formal treatment of objects with mutable fields and immutable methods, addition and deletion of computed properties from objects, nominal class-based object types, interfaces, structural object types with width-subtyping, and partial type erasure.

Formal core (§3). We develop SafeTS: a core calculus for Safe TypeScript. Our formalization includes the type system, compiler and runtime for a subset of Safe TypeScript, and also provides a dynamic semantics suitable for a core of both TypeScript and Safe TypeScript. Its metatheory establishes that well-typed SafeTS programs (with embedded runtime checks) simulate programs running under TypeScript's semantics (without runtime checks), except for the possibility of a failed runtime check that stops execution early (Theorem 1). Pragmatically, this enables programmers to switch between 'safe' and 'unsafe' mode while testing and debugging.

Full-fledged implementation for TypeScript (§4). Relying on differential subtyping and erasure, we extend SafeTS to the full Safe TypeScript language, adding support for several forms of inheritance for classes and interfaces; structural interfaces with recursion; support for JavaScript's primitive objects; auto-boxing; generic classes, interfaces and functions; arrays and dictionaries with mutability controls; enumerated types; objects with optional fields; variadic functions; and a simple module system. In all cases, we make use of a combination of static checks and RTTI-based runtime checks to ensure dynamic type safety.

Usability and Performance Evaluation (§5). We report on our experience using Safe TypeScript to type-check and safely compile about 120,000 lines of source code, including bootstrapping the Safe TypeScript compiler itself. In doing so, we found and corrected several errors that were manifested as type safety violations in the compiler and in a widely used benchmark. Quantitatively, we evaluate Safe TypeScript's tagging strategy against two alternatives, and find that differential subtyping (and, of course, erasure) offers significant performance benefits.

We conclude that large TypeScript projects can easily be ported to Safe TypeScript, thereby increasing the benefits of existing type annotations; that Safe TypeScript can reveal programming bugs both statically and dynamically; that statically typed code incurs negligible overhead; and that selective RTTI can ensure type safety with modest overhead.

We provide the full formal development and proofs of SafeTS, and more experimental details, in an accompanying technical report (Rastogi et al. 2014). Source code of our compiler and all our benchmarks are available at `https://github.com/nikswamy/SafeTypeScript`. We also provide an in-browser demo of Safe TypeScript in action at `http://research.microsoft.com/en-us/um/people/nswamy/Playground/TSSafe/`.

2. An overview of Safe TypeScript

Being sound, Safe TypeScript endows types with many of the properties that Java or C# programmers might expect but not find in TypeScript. On the other hand, Safe TypeScript is also intended to be compatible with JavaScript programming. As a language user, understanding what type safety means is critical. As a language designer, striking the right balance is tricky. We first summarize some important consequences of type safety in Safe TypeScript.

An object implements the methods in its type. Objects in JavaScript are used in two complementary styles. First, as mutable dictionaries, they use field-names to represent keys. Second, in a more object-oriented style, they expose methods to operate on their state. Safe TypeScript supports both styles. In less structured code, dictionary-like objects may be used: the type system ensures that fields have the expected type *when defined*. In more structured code, objects may expose their functionality using methods: the type system guarantees that an object always implement calls to the methods declared in its type, i.e., methods are *always defined* and *immutable*. The two styles can be freely mixed, i.e., a dictionary may have both methods and fields with functional types.

Values can be undefined. Whereas languages like C# and Java have one null-value included in all reference types, JavaScript has two: null and undefined. Safe TypeScript rationalizes this aspect of JavaScript's design, in effect removing null from well-typed programs while retaining only undefined. (Retaining only null is possible too, but less idiomatic.) For existing programs that may use null, our implementation provides an option to permit null to also be a member of every reference type. Note that undefined is also included in all native types, such as boolean and number. This supports, e.g., the pervasive use in JavaScript of undefined for false.

Type safety as a foundation for security. JavaScript provides a native notion of dynamic type safety. Although relatively weak, it is the basis of many dynamic security enforcement techniques. For instance, the inability to forge object references is the basis of capability-based security techniques (Miller et al. 2007). By compiling to JavaScript, Safe TypeScript (like TypeScript itself) still enjoys these properties. Moreover, Safe TypeScript provides higher level abstractions for encapsulation enforced with a combination of static and dynamic checks. For example, TypeScript provides syntax for classes with access qualifiers to mark certain fields as private, but does not enforce them, even in well-typed code. In §2.4, we show how encapsulations like private fields can be easily built (and relied upon!) in Safe TypeScript. Looking forward, Safe TypeScript's type safety should provide a useful basis for more advanced security-oriented program analyses.

Static safety and canonical forms. For well-typed program fragments that do not make use of the any type, Safe TypeScript ensures that no runtime checks are inserted in the code (although some RTTI may still be added). For code that uses only erased types, neither checks nor RTTI are added, ensuring that code runs at full speed. When adding RTTI, we are careful not to break JavaScript's underlying semantics, e.g., we preserve object identity. Additionally, programmers can rely on a canonical-form property. For example, if a value v is defined and has static type {ref:number}, then the programmer can conclude that v.ref contains a number (if defined) and that v.ref can be safely updated with a number. In contrast, approaches to gradual typing based on higher-order casts do not have this property. For example, in the system of Herman et al. (2010), a value r with static type ref number may in fact be another value wrapped with a runtime check—attempting to update r with a number may cause a dynamic type error.

In the remainder of this section, we illustrate the main features of Safe TypeScript using several small examples.

2.1 Nominal classes and structural interfaces

JavaScript widely relies on encodings of class-based object-oriented idioms into prototype-based objects. To this end, TypeScript provides syntactic support for declaring classes with single inheritance and multiple interfaces (resembling constructs in Java or C#) and its code generator desugars class declarations to prototypes using well-known techniques. Safe TypeScript retains TypeScript's classes and interfaces, with a few important differences illustrated on the code below:

```
interface Point { x:number; y:number }
class MovablePoint implements Point {
  constructor(public x:number, public y:number) {}
  public move(dx:number, dy:number) { this.x += dx; this.y += dy; }
}
function mustBeTrue(x:MovablePoint) {
  return !x || x instanceof MovablePoint;
}
```

The code defines a Point to be a pair of numbers representing its coordinates and a class MovablePoint with two public fields x and y (initialized to the arguments of the constructor) and a public move method. In TypeScript, all types are interpreted structurally: Point and MovablePoint are aliases for t_p ={x:number; y:number} and t_o ={x:number; y:number; move(dx:number, dy:number): void}, respectively. This structural treatment is pleasingly uniform, but it has some drawbacks. First, a purely structural view of class-based object types is incompatible with JavaScript's semantics. One might expect that every well-typed function call mustBeTrue(v) returns true. However, in TypeScript, this need not be the case. Structurally, taking v to be the object literal {x:0, y:0, move(dx:number, dy:number){}}, mustBeTrue(v) is well-typed, but v is not an instance of MovablePoint (which is decided by inspecting v's prototype) and the function returns false.

To fix this discrepancy, Safe TypeScript treats class-types nominally, but let them be viewed structurally. That is, MovablePoint is a subtype of both t_p and t_o, but neither t_p nor t_o are subtypes of MovablePoint. Interfaces in Safe TypeScript remain, by default, structural, i.e., Point is equivalent to t_p. In §4, we show how the programmer can override this default. Through the careful use of nominal types, both with classes and interfaces, programmers can build robust abstractions and, as we will see in later sections, minimize the overhead of RTTI and runtime checks.

2.2 A new style of efficient, RTTI-based gradual typing

Following TypeScript, Safe TypeScript includes a dynamic type any, which is a supertype of every non-erased type t. When a value of type t is passed to a context expecting an any (or vice versa), Safe TypeScript injects runtime checks on RTTI to ensure that all the t-invariants are enforced. The particular style of RTTI-based gradual typing developed for Safe TypeScript is reminiscent of prior proposals by Swamy et al. (2014) and Siek and Vitousek (2013), but makes important improvements over both. Whereas prior approaches require all heap-allocated values to be instrumented with RTTI (leading to a significant performance overhead, as discussed in §5), in Safe TypeScript RTTI is added to objects only as needed. Next, we illustrate the way this works in a few common cases.

The source program shown to the left of Figure 2 defines two types: Point and Circle, and three functions: copy, f and g. The function g passes its Circle-typed argument to f at the type any.

Clearly there is a latent type error in this code: in line 10, the function is expected to return a number, but circ.center is no longer a Point (since the assignment at line 7 mutates the circle and changes its type). Safe TypeScript cannot detect this error statically: the formal parameter q has type any and any property access on any-typed objects is permissible. However, Safe TypeScript does detect this error at runtime; the result of compilation is the instrumented code shown to the right of Figure 2.

```
1  interface Point { x:number; y:number }
2  interface Circle { center:Point; radius:number }
3  function copy(p:Point, q:Point) { q.x=p.x; q.y=p.y; }
4  function f(q:any) {
5      var c = q.center;
6      copy(c, {x:0, y:0});
7      q.center = {x:"bad"};  }
8  function g(circ:Circle) : number {
9      f(circ);
10     return circ.center.x;  }
```

```
1  RT.reg("Point",{"x":RT.num,"y":RT.num});
2  RT.reg("Circle",{"center":RT.mkRTTI("Point"), "radius":RT.num});
3  function copy(p, q) { q.x=p.x; q.y=p.y; }
4  function f(q) {
5      var c = RT.readField(q,"center");
6      copy(RT.checkAndTag(c, RT.mkRTTI("Point")),{x:0,y:0});
7      RT.writeField(q, "center", {x:"bad"});  }
8  function g(circ) {
9      f(RT.shallowTag(circ, RT.mkRTTI("Circle")));
10     return circ.center.x;  }
```

Figure 2: Sample source TypeScript program (left) and JavaScript emitted by the Safe TypeScript compiler (right)

As we aim for statically typed code to suffer no performance penalty, it must remain uninstrumented. As such, the copy function and the statically typed field accesses circ.center.x are compiled unchanged. The freshly allocated object literal {x:0,y:0} is inferred to have type Point and is also unchanged (in contrast to Swamy et al. (2014) and Siek and Vitousek (2013), who instrument *all* objects with RTTI). We insert checks only at the boundaries between static and dynamically typed code and within dynamically typed code, as detailed in the 4 steps below.

(1) Registering user-defined types with the runtime. The interface definitions in the source program (lines 1–2) are translated to calls to RT, the Safe TypeScript runtime library linked with every compiled program. Each call to RT.reg registers the runtime representation of a user-defined type.

(2) Tagging objects with RTTI to lock invariants. Safe TypeScript uses RTTI to express invariants that must be enforced at runtime. In our example, g passes circ:Circle to f, which uses it at an imprecise type (any); to express that circ must be treated as a Circle, even in dynamically typed code, before calling f in the generated code (line 9), circ is instrumented using the function RT.shallowTag whose implementation is shown (partially) below.

```
function shallowTag(c, t) {
  if (c!==undefined) { c.rtti = combine(c.rtti, t); }
  return c; }
```

The RTTI of an object is maintained in an additional field (here called rtti) of that object. An object's RTTI may evolve at runtime—Safe TypeScript guarantees that the RTTI decreases with respect to the subtyping relation, never becoming less precise as the program executes. At each call to shallowTag(c,t), Safe TypeScript ensures that c has type t, while after the call (if c is defined) the old RTTI of c is updated (using the function combine) to also recall that c has type t (Circle, in our example). Importantly for performance, shallowTag does not descend into the structure of c tagging objects recursively—a single tag at the outermost object suffices; nested objects need not be tagged with RTTI (a vital difference from prior work).

(3) Propagating invariants in dynamically typed code. Going back to our source program (line 5), the dynamically typed read q.center is rewritten to RT.readField(q,"center"), whose definition is shown (partially) below.

```
function readField(o,f) {
  if (f==="rtti") die("reserved name");
  return shallowTag(o[f], fieldType(o.rtti, f)); }
```

Reading a field f out of an object requires tagging the value stored in o.f with the invariants expected of that field by the enclosing object. In our example, we tag the object stored in q.center with RTTI indicating that it must remain a Point. The benefit we gain by not descending into the structure of an object in shallowTag is offset, in part, by the cost of propagating RTTI as the components of an object are accessed in dynamically typed code—empirically, we find that it is a good tradeoff (see §2.3 and §5).

(4) Establishing invariants by inspecting and updating RTTI. When passing c to copy (line 6), we need to check that c is a Point, as expected by copy. We do this by calling another runtime function RT.checkAndTag that (unlike shallowTag) descends into the structure of c, checks that c is structurally a Point and, if it succeeds, tags c with RTTI recording that it is a Point. We outline below the (partial) implementation of RT.checkAndTag with a simplified signature. In our example, where f is called only from g, the check succeeds.

```
function checkAndTag(v, t) {
  if (v === undefined) return v;
  if (isPrimitive(t)) {
    if (t === typeof v) return v;
    else die("Expected a " + t);
  } else if (isObject(t)) {
    for (var f in fields(t)) {
      checkAndTag(v[f.name], f.type);
    }; return shallowTag(v, t);
  } . . . }
```

Finally, we come to the type-altering assignment to q.center: it is instrumented using the RT.writeField function (at line 7 in the generated code), whose partial definition is shown below.

```
function writeField(o, f, v) {
  if (f==="rtti") die("reserved name");
  return (o[f]=checkAndTag(v,fieldType(o.rtti,f))); }
```

The call writeField(o, f, v) ensures that the value v being written into the f field of object o is consistent with the typing invariants expected of that field—these invariants are recorded in o's RTTI, specifically in fieldType(o.rtti, f). In our example, this call fails since {x:"bad"} cannot be typed as a Point.

2.3 Differential subtyping

Tagging objects can be costly, especially with no native support from JavaScript virtual machines. Prior work on RTTI-based gradual typing suggests tagging every object, as soon as it is allocated (cf. Siek and Vitousek 2013 and Swamy et al. 2014). Following their approach, our initial implementation of Safe TypeScript ensured that every object carries a tag. We defer a detailed quantitative comparison until §5.1 but, in summary, this variant can be 3 times slower than the technique we describe below.

Underlying our efficient tagging scheme is a new form of coercive subtyping, called differential subtyping. The main intuitions are as follows: (1) tagging is unnecessary for an object as long as it is used in compliance with the static type discipline; and (2) even if an object is used dynamically, its RTTI need not record a full description of the object's typing invariants: only those parts used outside of the static type discipline require tagging.

Armed with these intuitions, consider the program in Figure 3, which illustrates width subtyping. The triple of numbers p in toOrigin3d (a 3dPoint) is a subtype of the pair (a Point) expected by toOrigin, so the program is accepted and compiled to the code at the right of the figure. The only instrumentation occurs at the use of subtyping on the argument to toOrigin: using shallowTag, we tag p with RTTI that records just the z:number field—the RTTI need

```
function toOrigin(q:{x:number;y:number}) { q.x=0;q.y=0; }
function toOrigin3d(p:{x:number;y:number;z:number}) {
    toOrigin(p); p.z=0; }
toOrigin3d({x:17, y:0, z:42});
```

```
function toOrigin(q) { q.x=0; q.y=0; }
function toOrigin3d(p) {
    toOrigin(RT.shallowTag(p, {"z":RT.num})); p.z=0; }
toOrigin3d({x:17, y:0, z:42});
```

Figure 3: Width-subtyping in a source TypeScript program (left) and after compilation to JavaScript (right)

not mention x or y, since the static type of toOrigin's parameter guarantees that it will respect the type invariants of those fields. Of course, neglecting to tag the object with z:number would open the door to dynamic type safety violations, as in the previous section.

Differential width-subtyping. To decide what needs to be tagged on each use of subtyping, we define a three-place subtyping relation $t_1 <: t_2 \rightsquigarrow \delta$, which states that the type t_1 is more precise than t_2, and that δ is (to a first approximation) the difference in precision between t_1 and t_2. We let δ range over types, or $\emptyset$ when there is no loss in precision. We speak of t_1 as a δ-subtype of t_2. For width-subtyping on records, the relation includes the 'splitting' rule $\{\overline{x:t}; \overline{y:t'}\} <: \{\overline{x:t}\} \rightsquigarrow \{\overline{y:t'}\}$, since the loss between the two record types is precisely the omitted fields $\overline{y:t'}$. On the other hand, for a record type t, we have $t <:$ any $\rightsquigarrow t$, since in this case the loss is total. At each use of subtyping, the Safe TypeScript compiler computes δ, and tags objects with RTTI that record just δ, rather than the full type t_1.

Taking advantage of primitive RTTI. Our definition of differential subtyping is tailored specifically to the RTTI available in Safe TypeScript and, more generally, in JavaScript—on other platforms, the relation would likely be different. For primitive types such as numbers, for instance, the JavaScript runtime already provides primitive RTTI (typeof n evaluates to "number" for any number n) so there is no need for additional tagging. Thus, although number is more precise than any, we let number $<:$ any $\rightsquigarrow \emptyset$.

Similarly, for prototype-based objects, JavaScript provides an instanceof operator to test whether an object is an instance of a class (cf. §2.1); also a form of primitive RTTI. Hence, for a class-based object type C, we have $C <:$ any $\rightsquigarrow \emptyset$, meaning that Safe TypeScript does not tag when subtyping is used on a class-based object type. As such, our subtyping relation computes the loss between two types that may not already be captured by RTTI, hence subtyping is non-coercive on types with primitive RTTI.

Controlling differences to preserve object identity. Besides performance, differential subtyping helps us ensure that our instrumentation does not alter the semantics of TypeScript. Consider subtyping for function types. One might expect a value f:(x:Point)⇒3dPoint to be usable at type (x:3dPoint)⇒Point via a standard lifting of the width-subtyping relation to function types. Given that differential subtyping is coercive (tags must be added), the only way to lift the tagging coercions to functions is by inserting a wrapper. For example, we might coerce f to the function g below, which tags the z field of the argument and then tags that field again on the result.

```
function g(y) {
    var t = {"z":RT.Num};
    return shallowTag(f(shallowTag(y, t)),t); }
```

Unfortunately, for a language like TypeScript in which object identity is observable (functions are a special case of objects), coercive wrappers like the one above are inadmissible—the function g is not identical to f, the function it wraps. Our solution is to require that higher order subtyping only use $\emptyset$-*subtypes* for relating function arguments, i.e., only non-coercive subtyping is fully structural. Thus, we exclude (x:Point)⇒3dPoint $<:$ (x:3dPoint)⇒Point from the subtyping relation. Conversely, given $t =$ (x:t_1) ⇒t_2 and $t' =$ (x:t'_1) ⇒t'_2, we have $t <: t' \rightsquigarrow t$, when $t'_1 <: t_1 \rightsquigarrow \emptyset$ and $t_2 <: t'_2 \rightsquigarrow \emptyset$, since the $\emptyset$-difference ensures that no identity-breaking wrappers need to be inserted. Subtyping on functions is still coercive, however. In the relation above, notice that the difference is computed to be t, the type of the left-hand side. Thus, we coerce f : t to t' using shallowTag(f,t), which sets t in the rtti field of the function object f, thereby capturing the precise type of f in its RTTI.

2.4 A modality for type erasure

Selective tagging and differential subtyping enable objects with partial RTTI. Going further, it is useful to ensure that some objects *never* carry RTTI, both for performance and for modularity. To this end, Safe TypeScript introduces a new modality on types to account for RTTI-free objects.

User-controlled erasure. Consider a program that calls toOrigin3d on a large array of 3dPoints: the use of width-subtyping in the body of toOrigin3d causes every object in the array to get tagged with RTTI recording a z number field. This is wasteful, inasmuch as the usage of each 3dPoint is type-safe without any instrumentation. To avoid unnecessary tagging, the programmer may write instead:

```
function toOrigin(q:•Point) { q.x=0;q.y=0; }
```

The type •Point is the type of *erased* Points, i.e., objects that have number-typed fields x and y, but potentially no RTTI.[1] Subtyping towards erased types is non-coercive; i.e., 3dPoint<:•Point⇝ $\emptyset$. So, along one dimension, erased types provide greater flexibility, since they enable more subtyping at higher-order. However, without RTTI, the runtime system cannot enforce the typing invariants of •t values; so, along another dimension, erased types are more restrictive, since they exclude dynamic programming idioms (like field extension, reflection, or deletion) on values with erased types. In particular, •t is not a subtype of any. Balancing these tradeoffs requires some careful thinking from the programmer, but it can boost performance and does not compromise safety.

Information hiding with erased types. Since values of erased types respect a static type discipline, programmers can use erasure to enforce deeper invariants through information hiding. JavaScript provides just a single mechanism for information hiding: closures. Through the use of erased types, Safe TypeScript provides another, more idiomatic, form of hiding, illustrated below. Consider a monotonic counter object with a private v field hidden from its clients, and a public inc method for incrementing the counter.

```
var ctr: •{inc():number} = {v:0, inc(){ return ++this.v;} };
```

By introducing the newly allocated object ctr at an erased type, its client code is checked (statically and dynamically) to ensure conformance with its published API: only inc is accessible, not v. Without the erasure modality, clients could mutate the v field using statements like ctr["v"] $= -17$. We show an encoding of abstract types using erased types in §3.4 (Theorem 2).

Erasure, modularity, and trust. TypeScript programs rarely run in isolation; their environment includes APIs provided by primitive JavaScript arrays and strings, the web browser's document object model (DOM), JQuery, the file system for server-side JavaScript, etc. The default library used for typing TypeScript programs includes about 14,000 lines of specifications providing types to these

[1] §4 discusses how we fit erased types into TypeScript without any modifications to its concrete syntax; until then, we use the • to mark erased types.

external libraries, and even more comprehensive TypeScript library specifications are available online.[2] Recompiling all these libraries with Safe TypeScript is not feasible; besides, sometimes these libraries are not even authored in JavaScript. Nevertheless, being able to use these libraries according to their trusted specifications from within Safe TypeScript is crucial in practice.

The erasure modality can help: we mark such external libraries as providing objects with erased type, for two purposes: (1) since these external objects carry no RTTI, this ensures that their use within Safe TypeScript is statically checked for compliance with their specification; (2) the erased types ensure that their objects are never tagged—adding new fields to objects owned by external libraries is liable to cause those libraries to break.

As such, the type safety for typical Safe TypeScript programs is guaranteed only modulo the compliance of external libraries to their specifications. In scenarios where trust in external code poses an unacceptable risk, or when parts of the program need to carefully utilize features of JavaScript like eval that are outside our type-safe language, one might instead resort to the type-based isolation mechanism of TS* (Swamy et al. 2014). Specifically, TS* proposes the use of an abstract type Un to encapsulate untyped adversarial code and a family of type-directed coercions to safely manipulate Un-values. This mechanism is complementary to what Safe TypeScript offers. In fact, as discussed in §3.4, Safe TypeScript's erased types generalize their Un type. In particular, from Theorem 2, we have that the type $\bullet\{\}$ is the Safe TypeScript analog of Un.

Gradually evolving Safe TypeScript using erased types. Over the course of last year, as we were developing Safe TypeScript, TypeScript added several advanced features to its type system, notably generic types. Keeping pace with TypeScript is a challenge made easier through the use of erased types. Recall that values of erased types must be programmed with statically, since erased types are invisible to the runtime system. Adding TypeScript's generic class types to Safe TypeScript requires significant changes to the interplay between the Safe TypeScript compiler and its runtime. One way to minimize this upheaval is to add generic types to Safe TypeScript in two steps, first to the static type system only, where generics are well understood, and only thereafter to the runtime system, if needed. §4 explains how we completed the first step by treating all generic types as erased. This allows Safe TypeScript programmers to use generic types statically, promoting code reuse and reducing the need for (potentially expensive) subtyping. By restricting the interaction between generic types and any, the system remains simple and sound. So far, preventing the use of polymorphic values in dynamically typed code has not been a significant limitation.

3. SafeTS: the formal core of Safe TypeScript

SafeTS models a sizeable fragment of Safe TypeScript including erased types, primitive types, structural objects, and nominal classes and interfaces. In this section we define the SafeTS syntax, type system, and dynamic semantics. We model compilation as a translation from SafeTS to itself that inserts dynamic type checks. Due to space constraints, we cover only the subset of SafeTS without classes and interfaces; full details can be found in the technical report (Rastogi et al. 2014). The following section (§4) outlines the parts of SafeTS omitted here, and informally explains how our implementation extends SafeTS to all of Safe TypeScript.

Our main results are expressed as a weak forward-simulation (meaning that runtime checks do not alter the behavior of well-typed programs, except when a dynamic type safety violation is detected) and an information-hiding theorem (letting programmers build robust abstractions despite JavaScript's dynamic features).

[2] `https://github.com/borisyankov/DefinitelyTyped`

$$
\begin{array}{rlcl}
\text{Type} & \tau & ::= & t \mid \bullet\, t \\
\text{Dynamic type} & t & ::= & c \mid \mathsf{any} \mid \{M; F\} \mid \ldots \\
\text{Primitive} & c & ::= & \mathsf{void} \mid \mathsf{number} \mid \mathsf{string} \mid \mathsf{boolean} \\
\text{Method types} & M & ::= & \cdot \mid m(\overline{\tau_i}) : \tau \mid M_1; M_2 \\
\text{Field types} & F & ::= & \cdot \mid f{:}\tau \mid F_1; F_2 \\
\text{Expression} & e & ::= & v \mid \{\tilde{M}, \tilde{F}\} \mid e.f \mid e[e'] \\
 & & \mid & e.m(\overline{e_i}) \mid e[e'](\overline{e_i}) \mid \langle t \rangle e \mid \mathit{RT}(\overline{e \mid \tau}) \\
\text{Value} & v & ::= & \ell \mid x \mid \mathsf{c}_t \\
\text{Method defns.} & \tilde{M} & ::= & \cdot \mid m(\overline{x_i{:}\tau_i}) : \tau\ \{s; \mathsf{return}\ e\} \mid \tilde{M}_1, \tilde{M}_2 \\
\text{Field defns.} & \tilde{F} & ::= & \cdot \mid f :_\tau e \mid \tilde{F}_1, \tilde{F}_2 \\
\text{Statement} & s & ::= & e \mid \mathsf{skip} \mid s_1; s_2 \mid \mathsf{var}\ x{:}\tau = e \\
 & & \mid & x = e \mid e.f = e' \mid e[e'] = e''
\end{array}
$$

3.1 Syntax

The syntax for SafeTS is given above. We stratify types into those that may be used dynamically and those that may be erased. Dynamic types t include primitive types c, any, and structural object types $\{M; F\}$ where F is a sequence of field names f and their types τ and M is a sequence of method names m and their method types[3] written $(\overline{\tau_i}) : \tau$. As expected in TypeScript, methods also take an implicit this argument with the type of the enclosing object.

Like JavaScript, SafeTS separates expressions from statements. Expressions include values v, literals, static and dynamic field projections, static and dynamic method calls, and type casts. The form $\mathit{RT}(\overline{e \mid \tau})$ranges over the runtime functions (with several expression or type arguments) used to instrument compiled programs, modeled as primitives in SafeTS. As such, the *RT* form is excluded from source programs.

Values include memory locations ℓ, variables x (including the distinguished variable this), and literals, ranged over by the metavariable c_t. To reflect their primitive RTTI provided by JavaScript (and returned by typeof), we may subscript literals with their type, writing, e.g., $\mathsf{undefined}_{\mathsf{void}}$. Object literals are sequences of explicitly typed method and field definitions. (In our implementation, those types are first inferred by the TypeScript compiler, as shown in Figure 1.) Method definitions are written $m(\overline{x_i{:}\tau_i})\ :\ \tau\,\{s; \mathsf{return}\, e\}$. For simplicity, method bodies consist of a statement s and a return expression e; void-returning methods return undefined. Field definitions are written $f :_\tau e$ where τ is the type inferred by TypeScript and is not concrete syntax.

Statements include expressions, skip, sequences, typed variable definitions, variable assignments, and static & dynamic field assignments. Conditional statements and loops are trivial, so we relegate them to the full technical report.

Like TypeScript, SafeTS models functions as objects with a single method named call. Thus, functions in concrete syntax function $(\overline{x_i : t_i}){:}t\ \{s;\ \mathsf{return}\ e\}$ become $\{\mathsf{call}(\overline{x_i : t_i}){:}t\ \{s;\ \mathsf{return}\ e\}\}$ and function calls $e(\overline{e_i})$ become $e.\mathsf{call}(\overline{e_i})$.

3.2 Static semantics

Figure 4 presents a core of the static semantics of SafeTS. The main judgments involve the typing and compilation for expressions $\Gamma \vdash e : \tau \hookrightarrow e'$ and for statements $\Gamma \vdash s \hookrightarrow s'$ where the source and target terms are both included in SafeTS. (The embedding of compiled SafeTS to a formal semantics of JavaScript is beyond the scope of this work.)

The type system of SafeTS has two fragments: a fairly standard static type discipline that applies to most terms, and a more permissive discipline that applies to the more dynamic terms, such as dynamic field projection. Figure 4 gives the most interesting rules

[3] Although TypeScript provides different notation for methods and fields, it makes no semantic distinction between the two. In concrete syntax, the formal parameters in a method type must be named, although arguments are still resolved by position.

$$\boxed{\tau <: \tau' \leadsto \delta, \qquad \Gamma \vdash e : \tau \hookrightarrow e', \qquad \Gamma \vdash \tilde{M} \hookrightarrow \tilde{M}', \qquad \Gamma \vdash \tilde{F} \hookrightarrow \tilde{F}', \qquad \Gamma \vdash s \hookrightarrow s'}$$

$$\textsc{S-Refl}\ \frac{}{\tau <: \tau} \quad \textsc{S-Void}\ \frac{}{\textsf{void} <: \tau} \quad \textsc{S-PAny}\ \frac{}{c <: \textsf{any}} \quad \textsc{S-RAny}\ \frac{}{\{M;F\} <: \textsf{any} \leadsto \{M;F\}}$$

$$\textsc{S-Rec}\ \frac{\{M;F\} \neq \{M';F'\} \qquad F' \subseteq F \qquad \forall m(\overline{\tau_i}') : \tau' \in M'.\exists m(\overline{\tau_i}) : \tau \in M.\tau <: \tau' \wedge \forall i.\tau_i' <: \tau_i}{\{M;F\} <: \{M';F'\} \leadsto \{M \setminus M'; F \setminus F'\}} \quad \textsc{S-Erased}\ \frac{t <: t' \leadsto \delta}{[\bullet]t <: \bullet t'}$$

$$\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau \hookrightarrow x}\ \textsc{T-Env} \quad \frac{\tau = \mathit{sig}(\{\tilde{M},\tilde{F}\}) \qquad \Gamma, \textsf{this}{:}\tau \vdash \tilde{M} \hookrightarrow \tilde{M}' \qquad \Gamma \vdash \tilde{F} \hookrightarrow \tilde{F}'}{\Gamma \vdash \{\tilde{M},\tilde{F}\} : \tau \hookrightarrow \{\tilde{M}',\tilde{F}'\}}\ \textsc{T-Rec} \quad \frac{\Gamma \vdash e : \tau' \hookrightarrow e' \qquad \tau' <: \tau \leadsto \delta}{\Gamma \vdash f :_\tau e \hookrightarrow f :_\tau \textsf{shallowTag}(e',\delta)}\ \textsc{T-Fd}$$

$$\frac{\Gamma' = \Gamma, \overline{x_i{:}\tau_i}, \textsf{locals}(s) \qquad \Gamma' \vdash s \hookrightarrow s' \qquad \Gamma' \vdash e : \tau' \hookrightarrow e' \qquad \tau' <: \tau \leadsto \delta}{\Gamma \vdash m(\overline{x_i{:}\tau_i}) : \tau\ \{s; \textsf{return}\ e\} \hookrightarrow m(\overline{x_i{:}\tau_i}) : \tau\ \{s'; \textsf{return}\ \textsf{shallowTag}(e',\delta)\}}\ \textsc{T-Md} \quad \frac{\Gamma \vdash e : \tau' \hookrightarrow e' \qquad f{:}\tau \in \textsf{fields}(\tau')}{\Gamma \vdash e.f : \tau \hookrightarrow e'.f}\ \textsc{T-Rd}$$

$$\textsc{T-Call}\ \frac{\Gamma \vdash e : \tau \hookrightarrow e' \qquad m(\overline{\tau_i}) : \tau_r \in \textsf{methods}(\tau) \qquad \forall i.\Gamma \vdash e_i : \tau_i' \hookrightarrow e_i' \qquad \forall i.\tau_i' <: \tau_i \leadsto \delta_i}{\Gamma \vdash e.m(\overline{e_i}) : \tau_r \hookrightarrow e'.m(\textsf{shallowTag}(\overline{e_i}', \overline{\delta_i}))} \quad \textsc{T-WrX}\ \frac{\Gamma \vdash e : \tau' \hookrightarrow e' \qquad \tau' <: \Gamma(x) \leadsto \delta}{\Gamma \vdash x = e \hookrightarrow x = \textsf{shallowTag}(e',\delta)}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \hookrightarrow e_1' \qquad f{:}\tau \in \textsf{fields}(\tau_1) \qquad \Gamma \vdash e_2 : \tau_2 \hookrightarrow e_2' \qquad \tau_2 <: \tau \leadsto \delta}{\Gamma \vdash e_1.f = e_2 \hookrightarrow e_1'.f = \textsf{shallowTag}(e_2',\delta)}\ \textsc{T-Wr} \quad \frac{\forall j \in \{1,2,3\}.\Gamma \vdash e_j : t_j \hookrightarrow e_j'}{\Gamma \vdash e_1[e_2] = e_3 \hookrightarrow \textsf{write}(e_1', t_1, e_2', e_3', t_3)}\ \textsc{T-DWr}$$

$$\textsc{T-DRd}\ \frac{\forall j \in \{1,2\}.\Gamma \vdash e_j : t_j \hookrightarrow e_j'}{\Gamma \vdash e_1[e_2] : \textsf{any} \hookrightarrow \textsf{read}(e_1', t_1, e_2')} \quad \textsc{T-DCall}\ \frac{\forall j \in \{1,2,i\}.\Gamma \vdash e_j : t_j \hookrightarrow e_j'}{\Gamma \vdash e_1[e_2](\overline{e_i}) : \textsf{any} \hookrightarrow \textsf{invoke}(e_1', t_1, e_2', \overline{e_i}', \overline{t_i})} \quad \frac{\Gamma \vdash e : t' \hookrightarrow e'}{\Gamma \vdash \langle t \rangle e : t \hookrightarrow \textsf{checkAndTag}(e', t', t)}\ \textsc{T-C}$$

Figure 4: Typing and compiling a core of SafeTS, where $\delta ::= t \mid \emptyset$ and $\Gamma ::= \cdot \mid x{:}\tau \mid \Gamma, \Gamma'$ and $\tau <: \tau' \triangleq \tau <: \tau' \leadsto \emptyset$

from both fragments; we discuss them in turn. (Due to space constraints, we omit routine rules, such as those for typing literals.)

Differential subtyping is a ternary relation $\tau_1 <: \tau_2 \leadsto \delta$. As a shorthand, we write $\tau_1 <: \tau_2$ when $\delta = \emptyset$. Subtyping is reflexive (S-Refl) and is provably transitive. Via S-Void, the undefined value inhabits all types—in a stricter setting, one may choose to omit this rule. The rules S-PAny, S-RAny and S-Rec enforce any as a supertype for all primitive and object types, as well as subtyping on object types, as discussed in §2.3. Rule S-Erased stands for two rules: both t and $\bullet t$ are subtypes of $\bullet t'$, so long as t is a subtype of t'. Although there is a loss of precision (δ) when using this rule, uses of $\bullet t'$-terms have to be typed statically, so there is no need to add RTTI—hence the $\emptyset$-difference in the conclusion.

As we will see shortly, the use of subtyping when typing expressions and statements is carefully controlled—each use of subtyping may introduce a loss in precision which gets reflected into the RTTI of the compiled term by shallowTag(e, δ). Although not shown in the rules, when $\delta = \emptyset$ the call to shallowTag is optimized away.

Variables are typed and compiled to themselves using T-Env.

Objects are typed using T-Rec, by typing their method and field definitions in turn. The auxiliary function $\mathit{sig}(\{\tilde{M},\tilde{F}\})$ computes the type of the object itself for the this reference (discussed further in the next paragraph). Fields are typed using T-Fd: the initializer must be a subtype of the field type. The loss in precision δ due to the use of subtyping is reflected into the RTTI of e' using shallowTag. Methods are typed using T-Md. In the first premise, we extend Γ to contain not only the parameter bindings, but also bindings for local variables of the method body, denoted by locals(s). This models JavaScript's hoisting of local variables in method bodies. The rule then types s and the return expression e. The use of subtyping for the result is manifested in the compiled code by a call to shallowTag.

***Restricting the use of* this:** Foreshadowing the dynamic semantics, in a normal method call $v.m()$, the body of m executes with the implicit parameter this bound to v. For a function call $g()$, however, JavaScript's semantics for resolving the this-parameter is much more subtle—broadly speaking, the body of g executes with this bound to a global object. As such, relying on any properties of this in g is unsafe. We preclude the use of this in a non-method function $g = \textsf{call}(\overline{x_i{:}\tau_i}){:}\tau$ { s;return e } by typing it using this : $\bullet\{\}$, the type of an abstract reference to an object (see Theorem 2). Specifically, we define $\mathit{sig}(g) = \bullet\{\}$, whereas for all other objects $\mathit{sig}(\{\tilde{M},\tilde{F}\}) = \{M;F\}$, the point-wise erasure of method- and field-definitions to their types.

Field projections, method calls, local variable assignments, and field assignments are statically typed by T-Rd, T-Call, T-WrX, and T-Wr, respectively. These rules are routine apart from their use of shallowTag at each use of subtyping.

The dynamic fragment of SafeTS includes the rules T-DWr, T-DRd, T-DCall and T-C. In each case, we restrict the types of each sub-term involved to dynamic types t—erased types $\bullet t$ must respect the static discipline. When compiling the term, we generate a runtime check that mediates the dynamic operation in question, passing to the check the sub-terms and (some of) their static types as RTTI. In the next subsection, we discuss how each check makes use of RTTI to ensure dynamic type safety.

3.3 Dynamic semantics

Figure 5 presents selected rules from our small-step operational semantics, of the form $\mathbb{C} \longrightarrow \mathbb{C}'$ where each runtime configuration $\mathbb{C}$ is a pair of a state $\mathcal{C}$ and a program statement s. Our semantics models the execution of SafeTS programs both before and after compilation—the former is intended as a model of the dynamic semantics of a core of TypeScript, while the latter is a model of Safe TypeScript.

A state $\mathcal{C}$ is a quadruple $H; T; X; L$ consisting of a heap H mapping locations ℓ to mutable objects O and values v; a tag heap T mapping *some* of these locations to RTTI t (when executing source programs, the tag heap is always empty); a call stack X

where each element consists of a local store L and an evaluation context E; and a local store L mapping variables x to locations ℓ for the current statement. We use the notation $\mathcal{C}.H$ for the heap component of $\mathcal{C}$, $\mathcal{C} \lhd H$ for $\mathcal{C}$ with updated heap H, and use similar notations for the other components and also for $\mathbb{C}$.

Our runtime representation of objects includes a prototype field, a sequence of method definitions sharing a captured closure environment L, and a sequence of field definitions. For simplicity, we treat return e as a statement, although it can only occur at the end of a method body. Finally, we define evaluation contexts, E, as follows for both statements and expressions, embodying a strict left-to-right evaluation order.

$$\begin{array}{lll} E & ::= & \langle\rangle \mid E.f \mid E[e] \mid v[E] \mid RT(\overline{v|t}, E, \overline{e|t}) \mid \ldots \\ & \mid & E; s \mid \mathsf{var}\ x : t = E \mid \mathsf{return}\ E \mid \ldots \end{array}$$

Context rules. Figure 5 begins with E-DIE, where die is a distinguished literal that arises only from the failure of a runtime check; the failure bubbles up and terminates the execution. We omit the other, standard rules for evaluation contexts.

Field projection and update. Static field projection $\ell.f$ (E-RD) involves a prototype traversal using the *lookup* function, whose definition we omit. Dynamic field reads split into two cases; we show only the former: when an object reference ℓ' is used as a key into the fields of ℓ (E-DRD), as in JavaScript, ℓ' is coerced to a string by calling toString; when the key is a literal and H maps ℓ to an object, we return either its corresponding field, if any, or undefined. Dynamic field writes also have two cases; we show only the latter (E-DWR): we expect $\mathcal{C}.H(\ell)$ to contain an object, and we update its field f_{c} with v. (We write f_{c} for JavaScript's primitive coercion of a literal c to a field name.)

The calling convention and closures. E-DCALL shows a dynamic call of the method c in object ℓ. In the first premise, we use the auxiliary function *lookup_m_this* to traverse the prototype chain to find the method m and to implement JavaScript's semantics for resolving the implicit this argument to ℓ'. Usually $\ell = \ell'$, except when $m =$ call (i.e., a bare function call), when ℓ' defaults to a global object—this is safe since the type system ensures that functions do not use this in their bodies. Next, we gather all the local variables $\overline{y_j}$ from the method body s, and allocate slots for them and the function parameters in the heap. Locals and parameters are mutable (as shown in the next rule, E-WRX) and are shared across all closures that capture them, so we use one indirection and promote their contents to the heap. In the conclusion of E-DCALL, we push one stack frame, set the current local store to the captured closure environment (extended with the locals and parameters) and proceed to the method body. Dually, E-RET pops the stack and returns the value v to the suspended caller's context. Rule E-OBJ allocates objects: a fresh location in the heap is initialized with an object O whose prototype is set to a distinguished location $\hat{\ell}$, representing, concretely, Object.prototype in JavaScript. Initializing the methods involves capturing the current local store $\mathcal{C}.L$ as a closure environment. Initializing the fields is straightforward.

Two sources of RTTI for enforcing dynamic type safety. The main novelty of our dynamic semantics is in the remaining six rules, which enforce SafeTS's notion of dynamic type safety using RTTI. RTTI in SafeTS comes in two forms—there is *persistent* RTTI, associated with objects in the tag heap or available primitively on literals; and *instantaneous* RTTI, provided by the compiler among the arguments to the *RT* functions. The most precise view of an object's invariants available to a runtime check is obtained by combining both forms of RTTI, using the auxiliary partial function *combine* (Figure 6). An invariant of our system ensures that it is always possible to combine the persistent and instantaneous RTTI consistently, e.g., it is impossible for the tag heap to claim that an object has a field f : number while the instantaneous RTTI claims f : any. Additionally, our invariants ensure that the method types in the persistent RTTI are never less precise than the instantaneous method types—recall that any loss in precision due to subtyping on methods is recorded in the RTTI using shallowTag.

Reads and writes. E-READ mediates reading f_{c} from an object reference $\ell{:}t$; similarly, E-WRITE mediates writing $v{:}t'$ to f_{c} of $\ell{:}t$. In both cases, we combine any persistent RTTI stored at $T[\ell]$ (defined as $T(\ell)$ when $\ell \in dom(T)$ and $\{\cdot;\cdot\}$ otherwise) with t, the instantaneous RTTI of ℓ provided by the compiler, and then use the partial function *fieldType* to compute the type of f_{c}. If the field is present in the RTTI, we simply use its type t_f; unless the field name clashes with a known method name, the field type defaults to any; otherwise, *fieldType* is not defined, and both E-READ and E-WRITE are stuck—in this case, the configuration steps to die; we omit these routine rules. Given t_f, in E-READ, we project the field and then propagate t_f into the persistent RTTI of the value that is read before returning it. In E-WRITE, before updating f_{c}, we check that $v{:}t'$ is compatible with the expected type t_f.

Method and function invocations. E-INVM and E-INVF mediate these invocations. In E-INVM, the goal is to safely invoke method f_{c} on $\ell{:}t$ with parameters $\overline{v_i{:}t_i}$. If we find the method in ℓ's combined RTTI, we invoke it after checking that the parameters have the expected types, and then propagate the result type into the RTTI of the result. In E-INVF, the goal is to call a function-typed field of $\ell{:}t$. The handling is similar, except that instead of looking up a method, we traverse the prototype chain, project the field, and inspect that field's RTTI for a call signature. If we find the signature, we call the function just as in E-INVM. In both rules, if the method or function is not found, the configuration steps to die.

Propagating and checking tags. Finally we have two workhorses for the semantics: shallowTag (E-ST) and checkAndTag (E-CT). The semantics of the former is given by $[\![\mathsf{shallowTag}(v, \delta)]\!]_T$, an interpretation function on tag heaps. When $\delta = \emptyset$ or $v = \mathsf{c}$, there is no tag propagation and the function is the identity. On structural types, we use the *combine* function to update the tag heap—an invariant ensures that persistent RTTI evolves monotonically, i.e., it never gets less precise. Whereas shallowTag never fails, the interpretation $[\![\mathsf{checkAndTag}(v, t, t')]\!]_{T,H}$ is a partial function that either evaluates to a new tag heap or returns $\bot$. The interpretation is given by the function *ctaux*. The most interesting case involves checking whether ℓ can be given the type $\{M; F\}$. To do this, we consult $\{M'; F'\}$, the combined view of ℓ's RTTI, and if $\{M'; F'\} <: \{M; Fc\} \rightsquigarrow \delta$ where Fc are the fields shared between F and F', we tag ℓ with the loss in precision (if any); we then recursively checkAndTag ℓ's fields for each of the fields in F' not in F (F_{new}); and, if that succeeds, we finally propagate F_{new} to ℓ's RTTI. We prove that *ctaux* always terminates, even in the presence of cycles in the object graph.

3.4 Metatheory

Our main result is that compilation is a weak forward-simulation, which also implies subject reduction for well-typed programs. Our second theorem is an abstraction property for values of erased types, rewarding careful programmers with robust encapsulation. Our results apply to full SafeTS, including classes and nominal interfaces. In full SafeTS, runtime states $\mathcal{C}$ are *five*-tuples $S; H; T; X; L$ where the additional S is a signature that contains all class and interface declarations. The runtime functions like checkAndTag are parameterized by S as well, and use it to implement dynamic type safety for these constructs. Pragmatically, some runtime checks on classes can be more efficient, since they can be implemented primitively (e.g., using instanceof).

$$\textsc{E-Die}\ \frac{\mathcal{C}; s \longrightarrow \mathcal{C}'; \mathsf{die}}{\mathcal{C}; E\langle s\rangle \longrightarrow \mathcal{C}'; \mathsf{die}} \qquad \frac{\mathit{lookup}_{\mathcal{C}.H}(\ell, f) = v}{\mathcal{C}; \ell.f \longrightarrow \mathcal{C}; v}\ \textsc{E-Rd} \qquad \frac{}{\mathcal{C}; \ell[\ell'] \longrightarrow \mathcal{C}; \ell[\ell'.\mathsf{toString()}]}\ \textsc{E-Drd} \qquad \frac{H' = \mathcal{C}.H[\ell \mapsto \mathcal{C}.H(\ell)[f_{\mathsf{c}} \mapsto v]]}{\mathcal{C}; \ell[\mathsf{c}] = v \longrightarrow \mathcal{C} \triangleleft H'; v}\ \textsc{E-Dwr}$$

$$\frac{\mathit{lookup\_m\_this}_H(\ell, f_{\mathsf{c}}) = L'.m(\overline{x_i : \tau_i}) : \tau\{s\}, \ell' \qquad \mathsf{locals}(s) = \overline{y_j} \qquad \overline{\ell_j}, \overline{\ell_i} \text{ fresh} \qquad H' = H[\overline{\ell_i \mapsto v_i}, \overline{\ell_j \mapsto \mathsf{undefined}}]}{H; T; X; L; E\langle \ell[\mathsf{c}](\overline{v_i})\rangle \longrightarrow H'; T; (X; L.E); (L', \mathsf{this} \mapsto \ell', \overline{x_i \mapsto \ell_i}, \overline{y_j \mapsto \ell_j}); s}\ \textsc{E-Dcall}$$

$$\textsc{E-Wrx}\ \frac{H' = \mathcal{C}.H[\mathcal{C}.L(x) \mapsto v]}{\mathcal{C}; x = v \longrightarrow \mathcal{C} \triangleleft H'; \mathsf{skip}} \qquad \textsc{E-Ret}\ \frac{\mathcal{C}.X = X'; L.E}{\mathcal{C}; \mathsf{return}\ v \longrightarrow \mathcal{C} \triangleleft X'; L; E\langle v\rangle} \qquad \frac{\ell \text{ fresh} \quad H' = \mathcal{C}.H[\ell \mapsto \{\mathtt{proto}{:}\hat{\ell},\ \mathtt{m}{:}\mathcal{C}.L\,.\tilde{M},\ \mathtt{f}{:}\overline{f_i = v_i}\}]}{\mathcal{C}; \{\tilde{M}; \overline{f_i = v_i}\} \longrightarrow \mathcal{C} \triangleleft H'; \ell}\ \textsc{E-Obj}$$

$$\textsc{E-Read}\ \frac{t_f = \mathit{fieldType}(f_{\mathsf{c}}, \mathit{combine}(\mathcal{C}.T[\ell], t)) \qquad t_f <: \mathsf{any} \rightsquigarrow \delta}{\mathcal{C}; \mathsf{read}(\ell, t, \mathsf{c}) \longrightarrow \mathcal{C}; \mathsf{shallowTag}(\ell[\mathsf{c}], \delta)} \qquad \frac{t_f = \mathit{fieldType}(f_{\mathsf{c}}, \mathit{combine}(\mathcal{C}.T[\ell], t))}{\mathcal{C}; \mathsf{write}(\ell, t, \mathsf{c}, v, t') \longrightarrow \mathcal{C}; \ell[\mathsf{c}] = \mathsf{checkAndTag}(v, t', t_f)}\ \textsc{E-Write}$$

$$\frac{f_{\mathsf{c}}(\overline{t_i'}) : t' \in \mathsf{methods}(\mathit{combine}(\mathcal{C}.T[\ell], t)) \qquad t' <: \mathsf{any} \rightsquigarrow \delta}{\mathcal{C}; \mathsf{invoke}(\ell, t, \mathsf{c}, \overline{v_i}, \overline{t_i}) \longrightarrow \mathcal{C}; \mathsf{shallowTag}(\ell[\mathsf{c}](\overline{\mathsf{checkAndTag}(v_i, t_i, t_i')}), \delta)}\ \textsc{E-InvM} \qquad \frac{T' = [\![\mathsf{shallowTag}(v, \delta)]\!]_{\mathcal{C}.T}}{\mathcal{C}; \mathsf{shallowTag}(v, \delta) \longrightarrow \mathcal{C} \triangleleft T'; v}\ \textsc{E-St}$$

$$\frac{\mathit{lookup}_{\mathcal{C}.H}(\ell, f_{\mathsf{c}}) = \ell' \qquad \mathsf{call}(\overline{t_i'}) : t' \in \mathsf{methods}(\mathcal{C}.T[\ell']) \qquad t' <: \mathsf{any} \rightsquigarrow \delta}{\mathcal{C}; \mathsf{invoke}(\ell, t, \mathsf{c}, \overline{v_i}, \overline{t_i}) \longrightarrow \mathcal{C}; \mathsf{shallowTag}(\ell[\mathsf{c}](\overline{\mathsf{checkAndTag}(v_i, t_i, t_i')}), \delta)}\ \textsc{E-InvF} \qquad \frac{T' = [\![\mathsf{checkAndTag}(v, t, t')]\!]_{\mathcal{C}.T,\mathcal{C}.H}}{\mathcal{C}; \mathsf{checkAndTag}(v, t, t') \longrightarrow \mathcal{C} \triangleleft T'; v}\ \textsc{E-Ct}$$

Figure 5: Selected rules from SafeTS's dynamic semantics: $\mathcal{C}; s \longrightarrow \mathcal{C}'; s$

$$\begin{array}{lcl}
\mathit{fieldType}(f, t) & = & \text{if } f{:}t' \in \mathsf{fields}(t) \text{ then } t' \\
 & & \text{else if } f \notin \mathsf{methods}(t) \text{ then } \mathsf{any} \\
 & & \text{else } \bot \\
\mathit{combine}(t, t') & = & t \ \text{ if } t' = c \text{ or } t' = \mathsf{any} \\
\mathit{combine}(\{M; F\}, \{M'; F'\}) & = & \{M \cup \{m{:}\tau \in M' \mid m{:}\_ \notin M\}; \\
 & & F \uplus F'\} \\
[\![\mathsf{shallowTag}(v, \emptyset)]\!]_T & = & T \\
[\![\mathsf{shallowTag}(\mathsf{c}, \delta)]\!]_T & = & T \\
[\![\mathsf{shallowTag}(\ell, \{M; F\})]\!]_T & = & T[\ell \mapsto \mathit{combine}(T[\ell], \{M; F\})] \\
[\![\mathsf{checkAndTag}(v, t, t')]\!]_{T,H} & = & \mathit{ctaux}(v, t, t', T, H)
\end{array}$$

$$\begin{array}{lcl}
\mathit{ctaux}(\mathsf{undefined}, t, t', T, H) & = & T \\
\mathit{ctaux}(\mathsf{c}_c, t, c, T, H) & = & T \\
\mathit{ctaux}(v, t, \mathsf{any}, T, H) & = & T \\
\mathit{ctaux}(v, t, \bullet t', T, H) & = & \mathit{ctaux}(v, t, t', T, H) \\
\mathit{ctaux}(\ell, t, \{M; F\}, T, H) & = & \\
\quad \text{let } \{M'; F'\} = \mathit{combine}(T[l], t) & & \\
\quad \text{let } F_C = \{f{:}t \in F \mid f \in \mathit{dom}(F')\}, F_{\mathit{new}} = F \setminus F_C & & \\
\quad \text{check } (\{M'; F'\} <: \{M; F_C\} \rightsquigarrow \delta) & & \\
\quad \text{let } T_0 = [\![\mathsf{shallowTag}(\ell, \delta)]\!]_T \text{ and} & & \\
\quad \forall f_i{:}t_i \in F_{\mathit{new}}.T_i = \mathit{ctaux}(H[\ell][f_i], \mathsf{any}, t_i, T_{i-1}, H) & & \\
\quad [\![\mathsf{shallowTag}(\ell, \{\cdot; F_{\mathit{new}}\})]\!]_{T_n} & & \\
\mathit{ctaux}(v, t, t', T, H) & = & \bot \text{ if none of the above}
\end{array}$$

Figure 6: Auxiliary functions used in dynamic semantics

Compiling configurations. We extend the typing and compiling relation of Figure 4 to runtime configurations, writing $\mathbb{C} : \tau \hookrightarrow_\Sigma \mathbb{C}'$ where Σ is a heap-typing and τ is the type of the result of the stack of evaluation contexts. The main technicality is in the compilation of statements that include free heap-locations. In particular, when compiling $\mathcal{C}; s$ to $\mathcal{C}'; s'$, we relate the statements using a generalization of statement typing of the form $S; \Sigma; \mathcal{C}'.T; \Gamma \vdash s \hookrightarrow s'$, where Γ is derived from $\mathcal{C}.L$ and Σ. Intuitively, the heap typing Σ records the static type of a location at the instant it was allocated, while $\mathcal{C}'.T$ records the dynamic RTTI of a location, which evolves according to Definition 1 below.

To translate heap locations, we introduce the following rule:

$$\frac{\mathit{combine}_S(T[\ell], \Sigma(\ell)) = \mathit{combine}_S(T[\ell], t)}{S; \Sigma; T; \Gamma \vdash \ell : t \hookrightarrow \ell}\ \textsc{T-Loc}$$

This rule captures the essence of differential subtyping. In traditional systems with subtyping, we would type a location ℓ using any super-type of $\Sigma(\ell)$. With differential subtyping, however, any loss in precision due to subtyping must be reflected in the RTTI of ℓ, i.e., in $T[\ell]$. In T-Loc, we are trying to relate a source configuration $\mathcal{C}; \ell$ to a given target configuration $\mathcal{C}'; \ell$ (where $T = \mathcal{C}'.T$). So, we must pick a type t, such that the loss in precision in t relative to $\Sigma(\ell)$ is already captured in the persistent RTTI at $T[\ell]$. In fact, t may be more precise or even unrelated to $\Sigma(\ell)$, so long as, taken together with $T[\ell]$, there is no loss (or gain) in precision—the premise of T-Loc makes this intuition precise. Since *combine* is a partial function, the rule is applicable only when the persistent RTTI of ℓ is consistent with its static type.

The following relation constrains how tag heaps evolve. In particular, the information about a location ℓ never grows less precise. The auxiliary relation $\Sigma \sim T$ (in the full paper) states that $\Sigma(\ell)$ is consistent with $T(\ell)$ for each location ℓ in the domain of T, i.e., its static and dynamic types are never in contradiction.

Definition 1 (Tag heap evolution). *$\Sigma_1; T_1$ evolves to $\Sigma_2; T_2$, written $\Sigma_1; T_1 \rhd \Sigma_2; T_2$, when $\Sigma_2 \supseteq \Sigma_1$; $\Sigma_1 \sim T_1$; $\Sigma_2 \sim T_2$; and, for all $\ell \in \mathsf{dom}(T_2)$, we have either (1) $\ell \notin \mathsf{dom}(T_1)$, or (2) $T_1(\ell) = T_2(\ell)$, or (3) $T_1(\ell) = \{M_1; F_1\}$ and $T_2(\ell) = \{M_2; F_2\}$ with $M_1 \subseteq M_2$ and $F_1 \subseteq F_2$.*

Intuitively, our main theorem states that, if a source configuration $\mathbb{C}$ is typed at τ and compiled to $\mathbb{C}_1$, then every step taken by $\mathbb{C}$ is matched by one or more steps by $\mathbb{C}_1$, unless $\mathbb{C}_1$ detects a violation of dynamic type safety.

Theorem 1 (Forward Simulation). *If we have $\mathbb{C} : \tau \hookrightarrow_{\Sigma_1} \mathbb{C}_1$ then either both $\mathbb{C}$ and $\mathbb{C}_1$ are terminal; or, for some $\mathbb{C}'$ and $\mathbb{C}_1'$, we have $\mathbb{C} \longrightarrow \mathbb{C}'$, $\mathbb{C}_1 \longrightarrow^+ \mathbb{C}_1'$, and either $\mathbb{C}_1'.s = \mathsf{die}$ or for some $\Sigma_1' \supseteq \Sigma_1$ we have $\mathbb{C}' : \tau \hookrightarrow_{\Sigma_1'} \mathbb{C}_1'$ and $\Sigma_1; \mathbb{C}_1.T \rhd \Sigma_1'; \mathbb{C}_1'.T$.*

An immediate corollary of the theorem is the canonical forms property mentioned in §2. We can also read off the theorem a type safety property for target configurations, stated below, where $\Sigma \vdash \mathbb{C} : \tau$ abbreviates $\exists \mathbb{C}_0 . \mathbb{C}_0 : \tau \hookrightarrow_\Sigma \mathbb{C}$.

Corollary 1 (Type Safety). *If $\Sigma \vdash \mathbb{C} : \tau$ then either $\mathbb{C}$ is terminal or for some $\Sigma' \supseteq \Sigma$ we have $\mathbb{C} \longrightarrow^+ \mathbb{C}'$ and $\Sigma' \vdash \mathbb{C}' : \tau$.*

Information hiding. Our second theorem states that values with type $\bullet\{\}$ are immutable and perfectly secret in well-typed contexts. The theorem considers two well-typed configurations $\mathbb{C}_1$ and $\mathbb{C}_2$ that differ only in the contents of location $\ell{:}\bullet\{\}$ and shows that their reductions proceed in lock-step. It provides a baseline property on which to build more sophisticated, program-specific partial abstractions. For example, the monotonic counter from §2.4 chooses to allow the context to mutate it in a controlled manner and to reveal the result.

Theorem 2 (Abstraction of $\bullet\{\}$). *If $\Sigma(l) = \bullet\{\}$ and for $i \in \{1, 2\}$ we have $\Sigma \vdash \mathbb{C} \lhd H[\ell \mapsto O_i] : \tau$; then, for $n \geq 0$,*

$$\begin{array}{ll} \mathbb{C} \lhd H[\ell \mapsto O_1] \longrightarrow^n \mathbb{C}' \lhd H'[\ell \mapsto O_1] & \textit{if and only if} \\ \mathbb{C} \lhd H[\ell \mapsto O_2] \longrightarrow^n \mathbb{C}' \lhd H'[\ell \mapsto O_2]. & \end{array}$$

Theorem 2 also clarifies the similarity between type $\bullet\{\}$ and type Un from TS* (Swamy et al. 2014): both types are abstract to the context. However, erased types are more general. Type $\bullet$f:number contains more information (it has a field f with type number), whereas Un is an abstract type with no more information.

4. Scaling to Safe TypeScript

TypeScript has a multitude of features for practical programming and we adapt them all soundly for use in Safe TypeScript. Of particular interest are the many forms of polymorphism: inheritance for classes and interfaces, ad hoc subtyping with recursive interfaces, prototype-based JavaScript primitive objects, implicit conversions, ad hoc overloading, and even parametric polymorphism. Space constraints prevent a detailed treatment of all these features: we select a few representatives and sketch how SafeTS can be extended gracefully to handle them.

By restricting more advanced typing features (e.g., parametric polymorphism) to erased types, we improve the expressiveness of the static fragment of the language, while ensuring that these features do not complicate the (delicate) runtime invariants of Safe TypeScript and its interface with the type-checker.

4.1 Encoding type qualifiers

Since TypeScript does not syntactically support type qualifiers, we used a simple (though limited) encoding for the erasure modality. For instance, we give below the concrete syntax for the function toOrigin with erased types presented in §2.4.

```
module STS { interface Erased {} . . . } //Safe TypeScript library
// client code
interface ErasedPoint extends Point, STS.Erased {}
function toOrigin(q:ErasedPoint) { q.x=0;q.y=0; }
function toOrigin3d(p:3dPoint) { toOrigin(p); p.z = 0; }
```

To mark a type t as erased, we define a new interface I that extends both t and STS.Erased, a distinguished empty interface defined in the standard library. (We discuss inheritance of classes and interfaces in more detail, shortly.) In TypeScript, the type I has all the fields of t, and no others, so I is convertible to t. In Safe TypeScript, however, we interpret I (and, transitively, any type that extends STS.Erased) as an erased type. We use similar encodings to mark types as being immutable or nominal. While these encodings fit smoothly within TypeScript, they have obvious limitations, e.g., only named types can be qualified.

4.2 Inheritance

Class and interface extension. SafeTS provides a simple model of classes and interfaces—in particular, it has no support for inheritance. Adding inheritance is straightforward. As one would expect, since fields are mutable, classes and interfaces are not permitted to override inherited field types. Method overrides are permissible, as long as they respect the subtyping relation.[4] Specifically, when class C_1 extends C_0, we require that every overriding method m in C_1 be a subtype of the method that it overrides in C_0 (which in-turn enforces method arguments and return types to be related by $\emptyset$-subtyping). We refer to this as the *override-check*; it is analogous to rule S-REC in §3.

Implements clauses. Class inheritance in TypeScript is desugared directly to prototype-based inheritance in JavaScript. As an object may have only one prototype, multiple inheritance for classes is excluded. As in languages like Java or C#, a substitute for multiple inheritance is ad hoc subtyping, using classes that implement multiple interfaces. Unlike Java or C#, however, an instance of a class C can implicitly be viewed as an instance of a structurally-compatible interface I, even when C does not declare that it implements I. Nevertheless, in TypeScript, class declarations may be augmented with implements clauses mentioning one or more interfaces. For each such declaration, Safe TypeScript checks that the class provides every field and method declared in these interfaces, using the override-check above.

Extending $\emptyset$-subtyping with nominal interfaces. $\emptyset$-subtyping on the arguments and results of methods in the override-check can sometimes be too restrictive. As explained in §2.4, using erased types may help: their subtyping is non-coercive, since they need not carry RTTI. Dually, subtyping towards class- or primitive-types is also non-coercive, since their values always carry RTTI. Safe TypeScript makes use of implements-clauses to also provide non-coercive subtyping towards certain interfaces. By default, interface types are structural, but some of them can be qualified as *nominal*. Nominal interfaces are inhabited only by instances of classes specifically declared to implement those interfaces (as would be expected in Java or C#). More importantly, nominal interfaces are inhabited only by class instances with primitive RTTI, thereby enabling non-coercive subtyping and making S-REC and the overrides-check more permissive.

JavaScript's primitive object hierarchy. Aside from inheritance via classes and interfaces, we also capture the inheritance provided natively in JavaScript. Every object type (a subtype of $\{\}$) extends the nominal interface Object, the base of the JavaScript prototype chain that provides various methods (toString, hasOwnProperty, ...). Likewise, every function (an object with a call method in Safe TypeScript) extends the nominal interface Function. For instance, our subtyping relation includes $t <: \bullet\{\mathsf{toString}() : \mathsf{string}\} \rightsquigarrow \emptyset$.

Arbitrary prototype chains. Finally, we discuss a feature excluded from Safe TypeScript: programmers cannot build arbitrary prototype chains using JavaScript's __proto__ property, or using arbitrary functions as object constructors. The former (forbidden in the JavaScript standard, but implemented by several VMs) is prevented by treating __proto__ as a reserved property and forbidding its access both statically (where detectable) and at runtime. The latter is pre-

[4] TypeScript, more liberally, permits inheritance that overrides both fields and methods using an unsound *assignability* relation (Bierman et al. 2014).

vented by requiring that new be called only on objects with a constructor signature, only present on class types.

4.3 Generic interfaces, functions, and classes

The code below illustrates several valid uses of generic types in Safe TypeScript.

```
1   interface Pair⟨A,B⟩ { fst: A; snd: B }
2   function pair⟨A,B⟩(a:A,b:B): Pair⟨A,B⟩ {return { fst: a, snd: b }; }
3   declare var Array:{ new⟨A⟩(len:number):Array⟨A⟩; . . . }
4   interface Array⟨T⟩ {
5     push(...items:T []) : number; . . .
6     [key:number] : T }
7   class Map⟨A,B⟩ {
8     private map: Array⟨Pair⟨A, B⟩⟩;
9     constructor() { this.map = new Array(10); }
10    public insert(k:A,v:B) { this.map.push(pair(k,v)); } }
```

We have a declaration of a generic interface for pairs (line 1) and a generic function for constructing pairs (line 2), showing how types can be abstracted. Line 3 declares an external symbol Array (provided by the JavaScript runtime) at an implicitly erased type that includes a generic constructor—types can be abstracted at method signatures too. The constructor in Array builds a value of type Array⟨A⟩, an interface (partially defined at lines 4–6) that provides a push function, which receives a variable number of T-typed arguments, adds them all to the end of the array, and returns the new length of the array. The type Array⟨T⟩ also contains an *index signature* (line 6), which states that each Array⟨T⟩ is a map from number-typed keys to T-typed values, indicating that an array as:Array⟨T⟩ can be subscripted using a[i], for i:number. Finally, line 7 defines a generic class Map⟨A,B⟩.

Typing generics. Except for erasure (explained next), our static treatment of generic types is fairly straightforward: we extend the context with type variables, and allow type abstraction at interfaces, classes, and method/function boundaries. To enable instantiations of type variables at arbitrary types, including erased types, their subtyping only includes reflexivity (since erased types may not even be subtypes of any). We also support bounded quantification to extend subtyping for type variables. Type instantiations are inferred by TypeScript's inference algorithm, e.g., at line 9, TypeScript infers Pair⟨A,B⟩ for the arguments of new Array and, at line 10, Pair ⟨A,B⟩ and A, B for the arguments of push and pair, respectively.

Erasing generic types. To keep the interface between our compiler and runtime system simple, we erase all generic types, and we forbid subtyping from generic types to any. Take the pair function or the Array value, for example. Were we to allow it to be used at type any, several tricky issues arise. For instance, how to compute type instantiations when these values are used at type any? Conversely, should any be coercible to the type of pair? Ahmed et al. (2011) propose a solution based on dynamic seals, but it is not suitable here since dynamic seals would break object identity. Erasing generics types and forbidding their use in dynamically typed contexts sidesteps these issues. On the other hand, instances of generic interfaces need not always be erased. For example, Pair⟨number, string⟩ is a subtype of {fst:number; snd:string}, and vice versa. The latter type can be viewed structurally, and safely handled at type any, with the difference computed as usual. Thus, the erasure modality safely allows us to extend SafeTS with generics.

4.4 Arrays

TypeScript types arrays using the generic Array⟨T⟩ interface outlined in §4.3. Given their pervasive use, Safe TypeScript extends SafeTS with array types, written t[]. Arrays in JavaScript are instances of a primitive object called Array. However, all instances of arrays, regardless of their generic instantiation, share the same prototype Array.prototype. Thus, in contrast with Object and Function, we do not treat Array as a nominal interface type. Instead, we have t[] $<:$ any $\rightsquigarrow$ t[], meaning that array instances are tagged with RTTI as required by subtyping.

Further complications arise from subtyping. In TypeScript, array subtyping is both covariant (as in Java and C#) and contravariant, allowing for instance number[] $<:$ any[] $<:$ string[]. More conservatively, Safe TypeScript supports sound covariant subtyping for immutable arrays, based on a type CheckedArray in the standard library and a type qualifier for tracking immutability. Specifically, we have t[] $<:$ CheckedArray⟨s⟩$\rightsquigarrow$ t[] as long as t $<:$ s $\rightsquigarrow \emptyset$. The type CheckedArray includes only a subset of the methods of Array, for instance keeping map but excluding push. Additionally, the compiler inserts checks to prevent assignments on instances of CheckedArray. Finally, the runtime provides a function mutateArray ⟨S,T⟩(a:CheckedArray⟨S⟩): T[], which allows an immutable array a to be coerced back to an array with a different element type, after checking the RTTI of a for safety.

5. Experimental evaluation

We summarize below the experiments we conducted to measure the performance implications of our design choices, and to gain insight into how Safe TypeScript fares when used in practice.

1. We compared differential subtyping to a variant of Safe TypeScript that tags objects with RTTI as they are created. We find that RTTI-on-creation incurs a slow down by a factor of 1.4–3x.

2. We compared two implementation strategies for the tag heap: one using JavaScript's weak maps to maintain a global RTTI table "off to the side"; the other uses an additional field in tagged objects. We find the latter to be faster by a factor of 2.

3. To gain experience migrating from JavaScript, we ported six benchmarks from the Octane suite (`http://octane-benchmark.googlecode.com/`) to Safe TypeScript. We observe that, at least for these examples, migration is straightforward by initially typing the whole program using any. Even so, Safe TypeScript's variable scoping rules statically discovered a semantic bug in one of the benchmarks (navier-stokes), which has subsequently been fixed independently. For more static checking, we gradually added types to the ported benchmarks, and doing so also restored performance of the Safe TypeScript version to parity with the original JavaScript.

4. Finally, we gained significant experience with moving a large TypeScript codebase to Safe TypeScript. In particular, we migrated the 90KLOC Safe TypeScript compiler (including about 80KLOC from TypeScript-0.9.5) originally written in TypeScript. While doing so, Safe TypeScript reported 478 static type errors and 26 dynamic type errors. Once fixed, we were able to safely bootstrap Safe TypeScript—the cost of dynamic type safety is a performance slowdown of 15%.

5.1 Exploring the design space of tagging

Differential subtyping vs. RTTI-on-creation. Prior proposals for RTTI-based gradual typing suggest tagging every object (e.g. Swamy et al. 2014). We adapted this strategy to Safe TypeScript and implemented a version of the compiler, called STS*, that tags every object, array and function that has a *non-erased* type with RTTI upon creation. Thus STS* also benefits from the use of erased types, one of the main innovations of Safe TypeScript. In code that makes heavy use of class-based objects, Safe TypeScript and STS* have essentially the same characteristics—in both strategies, all class-based objects have RTTI (via their prototype chain) as soon as they are allocated. Finally, STS* has a few limitations, particularly when used with generic interfaces. Consider the function pair from §4.3. In Safe TypeScript, the function call pair(0,1) (correctly) allocates a pair v with no tags. Later, if we were to use subtyping and view v at type `any`, Safe TypeScript (again, correctly) tags v with the type {fst:`number`; snd:`number`}. In contrast, $\text{STS}^\star$ fails to allocate the correct RTTI for v, since doing so would require passing explicit type parameters to pair so as to tag it with the correct type at the allocation site. Thus, $\text{STS}^\star$ is not suitable for deployment, but it provides a conservative basis against which to measure the performance benefit of differential subtyping.

Object instrumentation vs. weak maps. Our default implementation strategy is to add a field to every object that carries RTTI. This strategy has some advantages as well as some drawbacks. On the plus side, accessing RTTI is fast since it is co-located with the object. However, we must ensure that well-typed code never accesses this additional field. To this end, we use a hard-to-guess, reserved field name, and we instrument the read, write, and invoke functions, as well as property enumerations, to exclude this reserved name. However, this strategy is brittle when objects with RTTI are passed to external, untrusted code. An alternative strategy to sidestep these problems makes use of WeakMap, a new primitive in the forthcoming ES6 standard for JavaScript already available in some (experimental) JavaScript environments: WeakMap provides a mapping from objects to values in which the keys are weakly held (i.e., they do not impede garbage collection). This allows us to associate RTTI with an object in state that is private to the Safe TypeScript runtime library. For class instances, we retain a field in the object's prototype. We refer to our implementation that use WeakMap as $\text{STS}^\dagger$.

The performance evaluation in the remainder of this section compares Safe TypeScript with its variants, $\text{STS}^\star$ and $\text{STS}^\dagger$, on Node.js-0.10.17, Windows 8.1, and an HP z-820 workstation.

5.2 Octane benchmarks

Octane is an open JavaScript benchmark suite. It contains 14 JavaScript programs, ranging from simple data structures and algorithms (like splay trees and ray-tracers) implemented in a few hundred lines, to large pieces of JavaScript code automatically generated by compilers, and even compilers implemented in JavaScript for compiling other languages to JavaScript.

The table alongside lists 6 programs we picked from the Octane benchmark, each a human-authored program a few hundred lines long which we could port to TypeScript with reasonable effort. All these programs use a JavaScript encoding of classes using prototypes. In porting them to Safe TypeScript, we reverted these encodings and used classes instead (since direct manipulation of prototype chains cannot be proven type-safe in Safe TypeScript); the 'classes' column indicates the number of classes we reverted; the number in parentheses indicates the number of abstract classes added while typechecking the program. We then added type annotations, primarily to establish type safety and recover good performance; the 'types' column indicates their number.

Name	LOC	classes	types
splay	394	2	15
navier-stokes	409	1(1)	41
richards	539	7(1)	30
deltablue	883	12	61
raytrace	904	14(1)	48
crypto	1531	8(1)	142

Without any type annotations, we pay a high cost for enforcing dynamic type safety. For the six unannotated Octane benchmarks, the slowdown spans a broad range from a factor of 2.4x (splay) to 72x (crypto), with an average of 22x. However, with the addition of types, we recover the lost performance—the slowdown for the typed versions is, on average only 6.5%. On benchmarks that make almost exclusive use of classes (e.g., raytrace and crypto) the performance of Safe TypeScript and $\text{STS}^\star$ is, as expected, the same.

In untyped code, the cost of additional tagging in $\text{STS}^\star$ is dwarfed by the large overhead of checks. However, in typed code, $\text{STS}^\star$ incurs an average slowdown of 66%, and sometimes as much as 3.6x. Finally, in dynamically typed code (involving many RTTI operations), $\text{STS}^\dagger$ is significantly slower than Safe TypeScript: 2x on average, parity in the best case, and 3.2x in the worst case. We have spent some effort on simple optimizations, mainly inlining runtime checks: this had a measurable impact on dynamically typed code, improving performance by 16% on average. However, there is still substantial room for applying many optimizations targeted towards detecting and erasing redundant checks.

We draw a few conclusions from our performance evaluation. First, differential subtyping is clearly preferable to RTTI-on-creation when trying to ensure good performance for statically typed code. Second, better type inference would significantly improve the experience of migrating from JavaScript to Safe TypeScript. Currently, we rely solely on TypeScript's support for local type inference within method bodies. Most of the annotations we added manually were for top-level functions and for uninitialized variables (where TypeScript defaults to inferring `any`). Inferring better types for these based on usage sites is left as future work. Whilst weak maps are an attractive implementation choice in principle, their performance overhead in $\text{STS}^\dagger$ is still too substantial for practical use, although as ES6 is more widely implemented, this option may become viable.

5.3 Bootstrapping Safe TypeScript

Our most substantial experience with Safe TypeScript to date has been with the Safe TypeScript compiler itself, which contains about 80 KLOC of code authored by the developers of TypeScript, and about 10 KLOC written by us. TypeScript supports an option (`--noImplicitAny`) that causes the compiler to report a warning if the type of any variable was inferred to be `any` without an explicit user annotation, and their compiler was developed with this option enabled. Thus, much of the code is carefully annotated with types.

Static error detection Bootstrapping the Safe TypeScript code base resulted in 478 static type errors. It took one author about 8 hours to diagnose and fix all these static errors, summarized below.

We detected 98 uses of bivariant subtyping of arrays: we fixed the covariant cases through the use of the immutable CheckedArray type (§4.4), and the contravariant cases by local code rewriting. Covariant subtyping of method arguments was observed 130 times, mostly in a single file that implemented a visitor pattern over an AST and due to binary methods in class inheritance. We fixed them all through the use of a runtime check. Variable scoping issues came up 128 times, which we fixed by manually hoisting variable declarations, and in 3 cases uncovering almost certain bugs on hard-to-reach code paths. Programmers confused methods and functions 52 times, e.g., projecting a method when passing a parameter to a higher-order function; which we fixed by local code rewriting. We lack the space to discuss the long tail of remaining error classes.

Dynamic type safety violations were detected 26 times, each a failed checkAndTag operation while running the compiler test suite. Five of these were due to attempted dynamic uses of covariant subtyping of mutable fields, primarily in code that was written by us—even when experts write code with type safety in mind, it is easy to make mistakes! Many failed downcasts (erased by TypeScript) were found in the existing code of TypeScript-0.9.5, which we fixed by rewriting the code slightly. Interestingly, two classes of dynamic type errors we discovered were in the new code we added. In order to implement Safe TypeScript, we had to reverse engineer some of the invariants of the TypeScript compiler. In some cases, we got this slightly wrong, expecting some values to be instances of a particular class, when they were not—the failed checks pointed directly to our mistakes. Another class of errors was related to a bug in the subtyping hierarchy we introduced while evolving the system with generic types, and which had not manifested itself earlier because of the lack of checked casts.

The performance overhead of safely bootstrapping the compiler (relative to bootstrapping the same code base with all runtime checks disabled) was a slowdown of only 15%. The added cost of runtime checks was easily paid for by the dozens of bugs found in heavily tested production code. We also bootstrapped the compiler using STS† and STS*, observing a further slowdown of 14% and 40%, respectively—the compiler makes heavy use of classes, for which we do not use weak maps or RTTI-on-creation, so the difference is noticeable, but not enormous.

We conclude that, at least during development and testing, opting in to Safe TypeScript's sound gradual type system can significantly improve code quality. For a code base that is already annotated with types (as most TypeScript developments are), the cost of migrating even a large codebase to Safe TypeScript can be reasonable: a day or two's worth of static error diagnosis followed by dynamic error detection with only slightly slower runtimes. On the other hand, to be fair, understanding the root cause of errors requires some familiarity with our type system, i.e., a developer interested in using Safe TypeScript effectively would probably have to understand (at least the informal parts of) this paper.

Experience with the TypeScript v1.1 compiler While we were working on Safe TypeScript, the TypeScript team released a new version, TypeScript-1.1, of the compiler. TypeScript-1.1 is comparatively much smaller (18K LOC) and faster (4x) (Turner 2014). It is a complete rewrite of the old compiler, including a design shift from being class-based to structural record- and interface-based. We have type-checked TypeScript-1.1 using Safe TypeScript, in the process providing inputs to the TypeScript team about the type errors found by Safe TypeScript. In preliminary experiments of using the instrumented version of TypeScript-1.1, we found that the overhead of the dynamic checks is much higher as compared to Safe TypeScript, which is based on TypeScript-0.9.5. The reason is that runtime checks are more expensive for structural types than for nominal class types. We are currently in the process of porting our type checker to TypeScript-1.1.

6. Related work

There has been considerable work on combining static and dynamic type-checking in a variety of language, too much to provide a comprehensive survey here—we discuss a few highlights.

Classic work on mixing static and dynamic types. Abadi et al. (1989) were among the first to study the semantics of a (simply) typed calculus with the explicit addition of a dynamic type. Around the same time, Fagan (1990) and Cartwright and Fagan (1991) considered adding static checks to a dynamically typed language. Their soft typing approach involved using a static type system to detect suspicious fragments of an unannotated program at compile time and guard them with runtime checks for dynamic type safety. Wright and Cartwright (1997) developed an implementation of soft typing for Scheme. Also related is the work of Henglein and Rehof (1995), who develop a system to translate a subset of Scheme to ML, while also handling polymorphism.

Static type systems for JavaScript. Given its ever-growing popularity over the last two decades, the earlier focus on LISP and Scheme has grown to include JavaScript. Early proposals for the addition of types to JavaScript were made by Thiemann (2005) who used singleton types and first-class record labels, and Anderson et al. (2005) who focused on type inference.

Guha et al. (2011) proposed a flow-sensitive static type system that refines types based on the control flow (`typeof` checks, comparison with `undefined`, etc.). The most recent incarnation of this line of work is TeJaS (Lerner et al. 2013), an extensible framework for exploring a range of static type systems for JavaScript. TeJaS features a sophisticated base type system (including bounded quantification, intersection and union types, mutable references, type-level functions, recursive types, object types, and a kind system). Despite this sophistication, or perhaps because of it, the base type system and its user-defined extensions come without a soundness argument.

Chugh et al. (2012) used both refinement and heap types to statically check some highly dynamic programming idioms of JavaScript: however, again, no soundness argument is given. Swamy et al. (2013) used an encoding of JavaScript (following a translation semantics by Guha et al. 2010) into a general-purpose, sound system of monadic refinement types to verify JavaScript safety.

Gradual typing. Siek and Taha (2006) popularized the term "gradual typing" for classifying systems that mix static and dynamic idioms while allowing the programmer to control their interaction using a language of type annotations and runtime checks. Tobin-Hochstadt and Felleisen (2006) introduced similar ideas almost simultaneously. The addition of runtime checks can cause hard-to-diagnose failures, particularly with higher-order types. To address this problem, Wadler and Findler (2009) present a notion of *blame* for gradual type systems, which allows failed runtime checks to identify the module in a program responsible for the failure. Blame may be less important in Safe TypeScript, inasmuch as it does not involve (identity-breaking) higher order wrappers, an important source of difficulty for error diagnosis.

Typed Scheme (Tobin-Hochstadt and Felleisen 2008) departs from the no-annotations, no-rejected-programs philosophy of Soft Scheme; it allows programmers to decorate their code with optional types and provides a control-flow sensitive type system to enforce them using a combination of static and dynamic checks. Takikawa et al. (2012) describe and formalize Typed Racket, the latest version of Typed Scheme. Their type system is similar in scope to Safe TypeScript in that it deals with methods, mutable fields, and structural subtyping. It also supports mixins and their subtle interaction with inheritance. It tracks flows of information between the typed and untyped spaces, relying on row polymorphism and (identity-breaking) sealed contracts to deal with mixins. It offers blame tracking should errors occur. Mixins in Typed Racket are reminiscent of dynamic field extension in Safe TypeScript; they may be coded in our system using a mixture of static types, runtime checks, and reflection.

Swamy et al. (2014) propose a gradual type system called TS* for JavaScript and utilize RTTI as a mechanism for ensuring type safety. As discussed in §1 and §2.4, the two systems are somewhat complementary. Safe TypeScript focuses on scale and flexibility; it incorporates new typing mechanisms leading to a more permissive type systems with lower runtime overhead. TS* focuses on isolation of untrusted code, and safe interoperability with code outside the fragment covered by Safe TypeScript. As an alternative to isolation, or in concert, one may also reduce trust in external code by resorting to the techniques of Feldthaus and Møller (2014), who develop tools to specifically check the correctness of a large TypeScript type definition repository.

Vitousek et al. (2014) present Reticulated Python, a framework for experimenting with gradual type systems in Python. They consider three cast semantics: (a) the well-known wrapper semantics, which breaks object identity; (b) a *transient semantics* that involves inserting runtime checks at call sites, function definitions, etc.; and (c) a *monotonic semantics* that locks objects with evolving field types. The transient semantics is not formalized and its soundness is unclear. The monotonic semantics is reminiscent of Swamy et al. (2014) and Safe TypeScript, except that types evolve with respect to an (intentionally) naïve subtyping hierarchy.

Allende et al. (2013) study three cast insertion strategies (placing casts at the call-site, the callee or a mix of the two) for Gradualtalk, a gradually-typed Smalltalk, and observe fluctuating performance when interacting with variously typed libraries. To mitigate this unexpected behavior, they recently present Confined Gradual Typing (Allende et al. 2014), an extension of a gradual typing with type qualifiers to track values between typed and untyped spaces. One of these qualifiers is similar to the erasure modality of Safe TypeScript, in that both exclude subtyping to `any`. However, their use of higher-order casts does not preserve object identity.

Wrigstad et al. (2010) advocate *like-types*, another mechanism for mixing static and dynamically typed code. They extend Thorn by allowing every (nominal) class type C to be qualified as Like$C$. Functions taking Like C parameters are statically checked against C's interface, whereas their callers are unconstrained. Hence, type safety involves dynamic checks on like-typed expressions. Richards et al. (2014) recently proposed like-types for JavaScript, building on TypeScript. Their model does not cover higher-order fields, method arguments, and references (their objects are purely functional). Our type system is more expressive, and our semantics more faithful to TypeScript. On the other hand, they use type information as a source of optimizations in the VM, an interesting aspect complementary to our work.

As discussed in §1, gradual type systems are starting to see industrial adoption. Microsoft's TypeScript has a gradual type system similar to the system of Siek and Taha (2007), but adds a number of unsound typing rules to support particular programming patterns, and erases all types during compilation, thereby excluding dynamic checks (Bierman et al. 2014). Google's Dart similarly relaxes soundness, and also normally compiles to JavaScript by erasing all types, but it has a checked mode that embeds ad-hoc runtime checks for early detection of some common errors. Facebook has released Hack, a gradual type system for PHP. Again, types are erased, and there is no soundness guarantee in the presence of `any` types.

7. Conclusions

Safe TypeScript is the first, large-scale, sound gradual type system for JavaScript. We have already argued for the benefits it provides to application developers, at a modest performance penalty. We also expect our work to be useful for researchers developing JavaScript program analyses, who may consider using Safe TypeScript as a baseline on which to build more sophisticated analyses. A large community effort continues to be expended on designing and implementing static analyses for JavaScript, and each must wrestle with the inherent dynamism of the language. Starting from Safe TypeScript may render static analysis problems more tractable, delegating to our compiler the task of reining JavaScript's dynamism and allowing the tool developer to focus on higher level properties.

Some challenges remain. Notably, JavaScript and TypeScript are moving targets: both languages evolve continuously, with the new standard of JavaScript (ES6) due in a few months and newer versions of TypeScript also expected. Whether or not the goodness of a sound type system will be embraced by the growing JavaScript and TypeScript community of developers, and evolve along with those languages, remains to be seen.

Acknowledgments Our thanks to Martín Abadi, Mads Torgersen and the TypeScript team for several useful discussions; to the anonymous reviewers of the paper and artifact for their valuable feedback; and to Matthias Felleisen, Shriram Krishnamurthi, and Sam Tobin-Hochstadt for suggesting improvements to our discussion of related work.

References

M. Abadi, L. Cardelli, B. Pierce, and G. Plotkin. Dynamic typing in a statically-typed language. In *Proceedings of POPL*, 1989.

A. Ahmed, R. B. Findler, J. G. Siek, and P. Wadler. Blame for all. In *Proceedings of POPL*, 2011.

E. Allende, J. Fabry, and É. Tanter. Cast insertion strategies for gradually-typed objects. In *Proceedings of DLS*, 2013.

E. Allende, J. Fabry, R. Garcia, and É. Tanter. Confined gradual typing. In *Proceedings of OOPSLA*, 2014.

C. Anderson, P. Giannini, and S. Drossopoulou. Towards type inference for JavaScript. In *Proceedings of ECOOP*, 2005.

G. Bierman, M. Abadi, and M. Torgersen. Understanding TypeScript. In *Proceedings of ECOOP*, 2014.

R. Cartwright and M. Fagan. Soft typing. In *Proceedings of PLDI*, 1991.

R. Chugh, D. Herman, and R. Jhala. Dependent types for JavaScript. In *Proceedings of OOSLA*, 2012.

M. Fagan. *Soft Typing: An Approach to Type Checking for Dynamically Typed Languages*. PhD thesis, 1990.

A. Feldthaus and A. Møller. Checking correctness of TypeScript interfaces for JavaScript libraries. In *Proceedings of OOPSLA*, 2014.

A. Guha, C. Saftoiu, and S. Krishnamurthi. The essence of JavaScript. In *Proceedings of ECOOP*, 2010.

A. Guha, C. Saftoiu, and S. Krisnamurthi. Typing local control and state using flow analysis. In *Proceedings of ESOP*, 2011.

F. Henglein and J. Rehof. Safe polymorphic type inference for Scheme: Translating Scheme to ML. In *Proceedings of FPCA*, 1995.

D. Herman, A. Tomb, and C. Flanagan. Space-efficient gradual typing. *Higher Order Symbol. Comput.*, 2010.

B. S. Lerner, J. G. Politz, A. Guha, and S. Krishnamurthi. TeJaS: Retrofitting type systems for JavaScript. In *Proceedings of DLS*, 2013.

Z. Luo. Coercive subtyping. *J. Log. Comput.*, 9(1):105–130, 1999.

M. S. Miller, M. Samuel, B. Laurie, I. Awad, and M. Stay. Caja: Safe active content in sanitized JavaScript. Unpublished paper, 2007.

A. Rastogi, N. Swamy, C. Fournet, G. Bierman, and P. Vekris. Safe & efficient gradual typing for TypeScript, MSR-TR-2014-99, 2014.

G. Richards, F. Z. Nardelli, C. Rouleau, and J. Vitek. Types you can count on. Unpublished paper, 2014.

J. G. Siek and W. Taha. Gradual typing for functional languages. In *Proceedings of Scheme and functional programming workshop*, 2006.

J. G. Siek and W. Taha. Gradual typing for objects. In *Proceedings of ECOOP*, 2007.

J. G. Siek and M. M. Vitousek. Monotonic references for gradual typing. Unpublished paper, 2013.

N. Swamy, J. Weinberger, C. Schlesinger, J. Chen, and B. Livshits. Verifying higher-order programs with the Dijkstra monad. In *Proceedings of PLDI*, 2013.

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.

A. Takikawa, T. S. Stickland, C. Dimoulas, S. Tobin-Hochstadt, and M. Felleisen. Gradual typing for first-class classes. In *Proceedings of OOPSLA*, 2012.

P. Thiemann. Towards a type systems for analyzing JavaScript programs. In *Proceedings of ESOP*, 2005.

N. Tillmann, M. Moskal, J. de Halleux, M. Fähndrich, and S. Burckhardt. Touchdevelop: app development on mobile devices. In *Proceedings of FSE*, 2012.

S. Tobin-Hochstadt and M. Felleisen. Interlanguage migration: from scripts to programs. In *Proceedings of DLS*, 2006.

S. Tobin-Hochstadt and M. Felleisen. The design and implementation of Typed Scheme. In *Proceedings of POPL*, 2008.

J. Turner. Announcing TypeScript 1.1 CTP, 2014. `http://blogs.msdn.com/b/typescript/archive/2014/10/06/announcing-typescript-1-1-ctp.aspx`.

M. M. Vitousek, A. M. Kent, J. G. Siek, and J. Baker. Design and evaluation of gradual typing for Python. In *Proceedings of DLS*, 2014.

P. Wadler and R. B. Findler. Well-typed programs can't be blamed. In *Proceedings of ESOP*, 2009.

A. K. Wright and R. Cartwright. A practical soft type system for scheme. *ACM Trans. Program. Lang. Syst.*, 19(1):87–152, Jan. 1997. .

T. Wrigstad, F. Z. Nardelli, S. Lebresne, J. Östlund, and J. Vitek. Integrating typed and untyped code in a scripting language. In *Proceedings of POPL*, 2010.