Anthony Tran
axt220073
CS 4337-503

“Safe and Efficient Gradual Typing for TypeScript” Article Critique

Introduction:

This article critique will be analyzing “Safe and Efficient Gradual Typing for TypeScript” by Aseem Rastogi, Nikhil Swamy, Cédric Fournet, Gavin Bierman, and Panagiotis Vekris. In their paper, Safe TypeScript is a new gradual type system implementation designed for Javascript. While there are pre-existing languages like Closure, Dart, or TypeScript, Safe TypeScript’s main objective is to maintain code integrity while also addressing modern issues such as scalability and code reusability. Additionally, the author discusses concepts such as “differential subtyping” and “erasure modality” in order to minimize the runtime of additional operations resulting from safety checks, while not compromising on the soundness of the runtime checks itself. However, even with the benefit of maintaining code integrity through a combination of static and dynamic checks, the cost of migrating from a large pre-existing codebase might accrue more complications than expected.

Summary:

When JavaScript was initially created, its main purpose was to act as a scripting language for small-scale web browsers. However, without natively supporting abstractions like static types and interfaces, JavaScript becomes increasingly harder to avoid runtime errors when dealing with large codebases. While modern languages like Dart attempt to fix this issue, they “remain intentionally unsound,” meaning they are easier to write, but their type annotation doesn’t prevent runtime errors. With Safe TypeScript, the regular Javascript check is followed by a runtime type information(RTTI) recompilation and a runtime check.

The first principle of runtime-type information compilation is partial erasure. Partial erasure splits variables into RTTI or erased types. RTTI types mark dynamic type variables while erased types are variables that are not subtypes of “any” variables. This allows modularity where erased-types allow objects created by external modules to not be marked by RTTI. Additionally, coders are able to control erasure to dictate if certain objects are marked by RTTI. This allows more forms of information hiding other than the default closures provided by JavaScript.

The second principle of runtime type information compilation is differential subtyping. Differential subtyping considers any objects marked with RTTI as a dynamic object. This allows the runtime check to be tailored towards specific parts of the program, rather than the entire program which allows more efficient checks. At the end, the sequence of static checks and RTTI runtime checks allows the program to maintain dynamic type safety.

At its core, Safe TypeScript is similar to other type-safe languages such as Java or C#, making it easier for programmers from statically-typed languages to interact with dynamically-typed JavaScript. To accomplish this, Safe TypeScript has to inherently support dynamic checks for JavaScript’s two main object styles: key-value dictionaries and object-oriented methods, while also accounting for the possible mixing of the two styles. To support JavaScript’s null and undefined data types, null was dropped and rationalized so that only undefined is kept. One problem however is deciding between whether to allocate more runtime to static or dynamic checks. The solution was to only implement runtime checks if there are dynamically allocated objects. If there are only erased types, RTTIs are not added, as their purpose is only to preserve Javascript semantics.

For scalability, the developers of Safe TypeScript focus on enforcing the different forms of Polymorphism, such as inheritance and parametric polymorphism. For example, in ad hoc subtyping, if a class is implementing more than one interface, Safe TypeScripts makes sure that each field and method passes an override-check.

In testing, Safe TypeScript was run through over a hundred thousand lines of code which allows it to take into account of bootstrapping and portability, meaning pre-existing JavaScript code could be converted into Safe TypeScript. In implementing Safe TypeScript runtime had a slowdown of fifteen percent. With codebases filled with dynamic types and classes, Safe TypeScript had a further slowdown of fifteen to twenty percent. However, in conclusion, it was found that migrating a codebase filled with type annotation from JavaScript to Safe TypeScript will only take a day or two with dynamic runtime detection.

Critique:

One of the strengths of the Safe TypeScript article is the conceptualization of runtime type information (RTTI). The author demonstrates how the architecture of Safe TypeScript allows for the combination of static and dynamic checks (eg. page 2). In the diagram, JavaScript parsing detects syntactical errors, followed by an STS type interface that checks for static errors. Afterward, a function call checks for RTTI-tagged objects in the dynamic check, then the new Safe TypeScript is emitted from the inputted JavaScript code.

On the other hand, a weakness of the article includes the poor statistical impact of Safe TypeScript. For example, “we measure a 15% runtime overhead for type safety” and “a further slowdown of 14%” can be a rather arbitrary value when dealing with different sizes of codebases. On a smaller-scale project, the performance overhead could be negligible, but with large modern codebases, the additional runtime from runtime checks can be substantially longer than the static check.

Additionally, the integration of previous codebases to Safe TypeScript would not be as easy as smaller projects. In “migrating even a large codebase to Safe TypeScript can be reasonable,” the author might be misleading as previous projects based on differing archetypes and semantics could cause more trouble than expected. This, followed by statistics on industry adoption of Safe TypeScript by NPM, could be implied that either Safe TypeScript’s design choice might be too differing from industry standards for migration to be considered, or the learning curve could be too unfamiliar for JavaScript developers, as it introduces ideas such as type erasure and qualifiers.

All in all, the strength of runtime type information on dynamic type check would be beneficial in maintaining code stability, however, the question lies in whether the cost of performing a Safe TypeScript dynamic check will be worth the price in time and resources.

Conclusion:

In summary, Safe TypeScript is a new type-safe tool designed to create a safe compilation environment for JavaScript code, while not forfeiting aspects of scalability and reusability as seen in modern languages such as Dart or Closure. To accomplish this, Safe TypeScript implements a static and dynamic check. Its dynamic check is composed of partial erasure and differential subtyping, where both elements use RTTI tags to dynamically check its objects. However, in implementing a type-safe environment, Safe TypeScript compromises on the length of its runtime when migrating from pre-existing codespaces. Having differing code semantics makes it complex to convert large codebases whilst maintaining a safe dynamic check. As a result, future research might take place in Safe TypeScript's integration with JavaScript codespaces or challenges for industry adoption, in order for Safe TypeScript to be more prevalent in today's code.

Citations:

1. Maryland, A. R. U. of *et al.* Safe & efficient gradual typing for typescript: Proceedings of the 42nd annual ACM Sigplan-SIGACT Symposium on principles of programming languages. *ACM Conferences* (2015). Available at: https://dl.acm.org/doi/10.1145/2676726.2676971. (Accessed: 11th November 2023)