Type Unsoundness in Practice: An Empirical Study of Dart

Gianluca Mezzetti Anders Møller Fabio Strocco
Aarhus University, Denmark
{mezzetti,amoeller,fstrocco}@cs.au.dk

Abstract

The type system in the Dart programming language is deliberately designed to be unsound: for a number of reasons, it may happen that a program encounters type errors at runtime although the static type checker reports no warnings. According to the language designers, this ensures a pragmatic balance between the ability to catch bugs statically and allowing a flexible programming style without burdening the programmer with a lot of spurious type warnings.

In this work, we attempt to experimentally validate these design choices. Through an empirical evaluation based on open source programs written in Dart totaling 2.4 M LOC, we explore how alternative, more sound choices affect the type warnings being produced. Our results show that some, but not all, sources of unsoundness can be justified. In particular, we find that unsoundness caused by bivariant function subtyping and method overriding does not seem to help programmers. Such information may be useful when designing future versions of the language or entirely new languages.

Categories and Subject Descriptors D.3.3 [*Language Constructs and Features*]

Keywords type systems; language design; gradual typing

1. Introduction

One of the greatest controversies in the history of programming language design has been whether or not to use static type checking. Many mainstream languages, such as Java and C++, are based on static type checking, but dynamically typed languages, such as JavaScript and Python, have become immensely popular and widespread. Proponents of static typing argue that it helps detecting errors early and that it provides a strong foundation for IDE functionality, while others argue that dynamic typing gives more flexibility and increases programmer productivity [19, 33]. Within the last decade, many proposals have been made to combine static and dynamic typing, aiming to achieve the best of the two worlds. A promising idea is *optional typing* [2], including gradual typing [27], which allows programmers to use type annotations selectively and thereby control which parts of a program are type checked statically. Many language designers have picked up on this idea by adding optional type annotations and gradual typing to existing dynamically typed languages. Examples include TypeScript (for JavaScript) [21, 26], Hack (for PHP) [11], Reticulated Python [34], Typed Scheme/Racket [32], and Typed Lua [17].

The Dart language supports a variant of gradual typing, not retrofitted into an existing statically or dynamically typed language, but built-in from the very first version. The Dart language and infrastructure is being developed by Google and standardized by Ecma [7]. It was announced in 2011 and has become quite popular: Google AdWords is implemented in Dart, and a search on GitHub quickly finds millions of lines of Dart code. The Dart language designers thoughtfully made some rather controversial choices regarding the type system. One of the key principles has been that the type system should not get in the way of the programmer, and this has been more important than soundness [4]. The type system in Dart has been designed to catch typical errors at compile time, but it does not give guarantees: it is possible that a fully type annotated program passes static type checking without any warnings, yet the program may fail with a type error at runtime. The language designers argue that this pragmatic choice is good both for the language implementors and the programmers who use the language. As an example, the language specification briefly motivates the choice of unsound variance of generic types [7, Section 19]:

> *Experience has shown that sound type rules for generics fly in the face of programmer intuition.*

A recent introductory book on Dart, written by Gilad Bracha who is the main editor of the Dart Ecma standard specification, contains a foreword by Erik Meijer that also advocates for unsoundness [3]:

DLS '16, October 30–November 4 2016, Delft, Netherlands

ACM 978-1-4503-4445-6/16/10…$15.00
DOI: http://dx.doi.org/10.1145/10.1145/2989225.2989227

Type systems are often highly non-linear and after a certain point their complexity explodes, while adding very little value to the developer and making life miserable for the language implementor. [...] While theoretically unsound, unsafe variance actually feels rather natural for most developers, and I applaud the choice the Dart designers made here. [...] languages that have chosen in favor of static type safety for generics do so at the expense of their users.

Type system unsoundness is not as outrageous as some theoretically inclined programming language researchers may think. Also in other languages, runtime type checks are often used to complement corners of a static type system. For example, in Java and C#, type casts allow the programmer to overrule the static type system, which may result in runtime cast errors, and array write operations are subjected to runtime type checks because of the design choice of allowing covariant subtyping for array types. Dart takes this further by allowing more programs to type check, aiming to issue static type warnings only in situations that are likely to indicate errors, at the cost of having more unsoundness.

The potential drawbacks of unsoundness are well known. Unsoundness in the type system may cause subtle errors to remain unnoticed until the programs are put into use, and it makes it difficult to utilize type information for program optimization purposes. This raises the question of whether the advantages of the various sources of unsoundness in Dart's type system outweigh the drawbacks.

To our knowledge there is no empirical evidence of the positive value of unsoundness for the programmers who use Dart. The various sources of unsoundness are informally justified by examples, typically showing scenarios where sound alternatives would result in type warnings that do not indicate actual bugs and are difficult to circumvent [4]. However, it is not clear to what extent such scenarios are representative of realistic programs.

The Dart language designers could indeed have chosen a "more sound" type system. In recent work, Ernst et al. [9] have categorized the sources of Dart's unsoundness and presented two possible alternative designs: one that is fully sound akin to e.g. Java (with runtime errors being possible at type casts), and one that rules out all message-not-understood errors in fully annotated code while still permitting subtype-violation errors at runtime. Many other variations are of course possible. This aligns well with Bracha's principle of pluggable type systems [2]. Notice that subtyping is used both by the static type checker and at runtime in checked-mode execution. Providing alternative static type checkers gives the programmers more options, but it may not be desirable to also change which runtime type checks are conducted.

In this work, we attempt to provide a posteriori empirical justification of the various sources of unsoundness in Dart's type system, based on a study of Dart programs available in the 'pub' repository, totaling 2.4 M LOC. Empirical justification of language design choices is unfortunately rare (we discuss some exceptions in Section 5); it is our hope that our study can not only explore this specific aspect of Dart but also give inspiration to further experimental studies of language design choices.

Hypothesis and Methodology

As observed by Ernst et al. [9], for each source of unsoundness in Dart's type system, there is a natural choice for an alternative sound design. For example, a natural sound alternative to covariant generics is to use invariant generics. (In Section 3 we recapitulate these categories of unsoundness and the sound alternatives.) This gives us a spectrum between the unsound standard type system and a fully sound alternative: for each category we may choose either the unsound or the sound variant. We can then test the following main hypothesis, aiming to validate the choices made by the designers of Dart:

> *For each source of unsoundness, switching to the sound alternative would result in a significant increase in the number of warnings raised by the static type checker or runtime type errors in checked-mode execution of realistic Dart programs, and without causing a significant increase in the number of programmer mistakes being caught by the type checker.*

Confirming this hypothesis would legitimate the Dart design choices on an empirical basis. Conversely, in case the hypothesis turns out to be rejected for one or more sources of unsoundness, we can conclude that the programmers gain little or no benefit from those sources of unsoundness, which may be useful information when developing future versions of the language, or for the design of new languages.

Testing the hypothesis is, however, not trivial. As a first step, we implement the modifications of the type checker and the runtime system, such that we can see how they affect the type warnings and runtime errors on the available collection of Dart programs. In case new warnings or errors do appear, we conduct a manual study of a subset of them to determine whether they typically are mistakes that the programmer likely would want to fix or they are artifacts of the type system, that is, situations where the program works as intended and the programmer would likely have a struggle to please the type system. This is admittedly a subjective decision; for example, type warnings may appear in code that technically works fine but maybe is questionable from a maintainability perspective.

The focus of our study is on the type system from the point of view of programmers who use Dart. Nevertheless, we also find out how much effort is required to implement the more sound variations, to see whether they indeed make "life miserable for the language implementor" (cf. the quote from Meijer).

Contributions

In summary, the contributions of this paper are as follows.

- We present a methodology for experimentally evaluating the pros and cons of the various sources of unsoundness in Dart's type system. To our knowledge, no such evaluation has been done before for any language with an (intentionally) unsound type system.
- We use Ernst et al. [9] as a starting point for classifying the sources of unsoundness, but to better study the effect of each of them in isolation, we suggest a more fine-grained characterization of the sound alternatives. As part of this effort, we also clarify the connections between Dart and gradual typing [27, 28].
- We report on our implementation of a modified type checker and runtime system, which provides a sound alternative for each source of unsoundness. The implementation is straightforward and involves only 121 LOC in the type checker and 42 LOC in the runtime system, which shows that all the modifications require only little effort for the language implementors.
- We conduct experiments on 1 888 real-world Dart programs, thereof 390 with functioning test suites, to evaluate how switching to the sound alternatives affects the type checker warnings and runtime type errors. This allows us to connect each new warning and error to a source of unsoundness. In addition, we report on a preliminary manual study of a small subset of the warnings to determine whether they typically indicate mistakes that the programmer likely would want to fix or they are artifacts produced by a fastidious type system.
- Our main finding is that unsoundness caused by bivariant function subtyping and method overriding is little used by programmers. Switching to a sound alternative causes only few more type warnings, and most of those warnings are not just type system artifacts but indicate programming mistakes.

In Section 2 we briefly explain how types work in Dart and discuss the notion of soundness. In Section 3 we describe the various sources of unsoundness in Dart and explore sound alternatives. Section 4 explains our experimental setup for testing our hypothesis and discusses our findings. Section 5 reviews related work, and Section 6 concludes.

2. Types and Dart

The Dart programming language supports both object-oriented and functional programming [7]. We briefly summarize the main aspects of its type system, which is the subject of our study. A formalization of the core of the language with a focus on the type system can be found in the recent work by Ernst et al. [9].

Every runtime value is an object, and every object is an instance of a class. This includes primitive values, functions, and `null`. Type annotations are optional, which enables a form of gradual typing [27, 28]. The default type, `dynamic`, effectively disables static type checking for the variable or field in question. The language is memory safe (unlike e.g. C), so the runtime types of values cannot be circumvented. Types are nominal (unlike other languages with optional typing [34]), except function types which are structural. We write $T <: S$ if T is a subtype of S, and $T <:> S$ is an abbreviation of $S <: T \ \vee \ T <: S$. Classes can be generic, and type parameters are reified (unlike e.g. generics in Java). Generic methods and functions are not currently part of the Dart language specification.[1]

Dart distinguishes between type *warnings*, which are reported by the static type checker, and type *errors*, which may happen at runtime. A program is *well typed* if no warnings are produced by the type checker. Different kinds of runtime type errors can occur, here using the terminology of Ernst et al. [9]: a *message-not-understood* (MNU) error occurs if attempting to access a field or method that does not exist (excluding null pointer dereferences); a *subtype-violation* (SV) error occurs if a value does not match the declared (i.e., static) type at a write operation. In this work we also consider a third kind of error: a *read-subtype-violation* (RSV) error occurs if the runtime type of the value being read from a variable or field or returned from a method call is not a subtype of the declared type. Programs can be executed in *production mode* or *checked mode*; subtype-violation checks are only performed in checked mode. Except for the subtype-violation checks, type annotations have no effect at runtime; in particular, non-well-typed programs can be executed.

Soundness of type systems To be able to explain the various alternatives to Dart's type system in the following section, it is helpful to recall the terminology by Cardelli [5]. In general, type systems are concerned with two kinds of runtime errors: a *trapped* error is one that causes the computation to "stop immediately"; an *untrapped* error instead can "go unnoticed (for a while) and later cause arbitrary behavior". As Dart is memory safe, untrapped errors cannot cause complete havoc, but they can result in MNU and SV errors later in the execution. In Dart, MNU errors are trapped, whereas SV errors are trapped only in checked mode but untrapped in production mode, and RSV errors are generally untrapped (but some are prevented by the SV checks). For example, the following program erroneously assigns a number to a variable of type `String`:

```
1 String x = 5;
2 print(x.length);
```

In checked mode, the error causes the program to stop in line 1 due to the SV check, while in production mode, the error is unnoticed until line 2 where it causes the program to stop with an MNU error. Untrapped errors are often more problematic than trapped ones because of the possibly large

[1] `https://github.com/leafpetersen/dep-generic-methods`

distance between where an error occurs and where it is detected. For the remainder of this paper, we focus on checked mode, and we shall later see examples of RSV errors. Interestingly, absence of SV errors does not imply absence of RSV errors.

The purpose of a type system is to check statically whether a given category of errors, called *forbidden* errors, can occur at runtime [5]. A type system is *sound* with respect to a given collection of forbidden errors if those errors cannot occur in well-typed programs. In this paper, unless otherwise noted, we select MNU and SV as the forbidden errors, while still allowing RSV errors, null pointer errors, type cast errors (i.e., errors triggered by the `as` operator), and array-out-of-bounds errors.

Note that some of the variations of the type system that we present involve modifying the subtyping relation, which affects what errors are considered forbidden and therefore also the meaning of soundness.

A *false positive* is a spurious warning from the type checker about a potential forbidden error that cannot occur at runtime. Conversely, a *false negative* is a runtime error that is not reported by the type checker; such errors only occur if the type system is unsound. With our main hypothesis in mind, we are interested not only in soundness but also in whether or not warnings indicate bugs. We informally distinguish between warnings that point to *programmer mistakes*, i.e. issues the programmer likely would want to fix, and *artifacts of the type system*, which are warnings that arise in program code that works as intended and where the programmer would likely have a struggle to please the type system if the warning should be avoided. Note that type system artifacts do not have to be false positives, nor vice versa.

Gradually typed soundness Soundness in the sense defined above is clearly lost when optional typing (or type `dynamic`) is introduced. The literature on gradual typing provides a notion of soundness that is suitable for this setting. The term "gradual typing" is sometimes used as a synonym for "optional typing" although the original work on gradual typing had a different intention. To clarify the terminology, Siek et al. [28] have recently proposed a set of requirements a language should satisfy in order to be called gradually typed. These include the following *gradually typed soundness* criteria, here stated informally and adapted to a Dart setting:[2]

1. A well-typed fully annotated program (not containing `dynamic`) cannot encounter MNU nor SV errors[3] at runtime (i.e., the type system is sound for fully annotated programs).
2. A well-typed program (which may contain `dynamic`) cannot encounter MNU or RSV errors in checked-mode execution (but SV errors could be ubiquitous).

Dart's type system is unsound: there exist well-typed programs that have MNU and SV errors. What is more, Dart violates both the criteria for gradually typed soundness mentioned above, for reasons explained in the following.

3. Sources of Unsoundness in Dart

Ernst et al. [9] have suggested two sound variants of the type system, one that treats both MNU and SV as forbidden errors, and one that forbids MNU but not SV. (The latter variant is called *message safety*.) In this section we review the sources of unsoundness, and for each of them, present a sound alternative. Unlike Ernst et al. we want to study the consequences of each source of unsoundness in isolation, which in some cases requires a slightly more fine-grained characterization of the sound alternatives.

A central part of the type system is the subtype relation, which is used both for static type checking and for runtime type checks. It is technically possible to use different notions of subtyping for these two purposes, but to preserve comprehensibility, each variant of the type system we discuss uses the same subtype relation for type checking and at runtime.

The sources of unsoundness can be grouped into the following 10 categories:

- Type `dynamic`
  - at lookups (**DLOOK**)
  - with ground types (**DGND**)
  - with generic types (**DGEN**)
  - with function types (**DFUN**)
- Symmetric assignability (**SYA**)
- Covariant generics (**COGE**)
- Bivariant function subtyping
  - at parameter types (**BIFSP**)
  - at return types (**BIFSR**)
- Bivariant method overriding
  - at parameter types (**BIMOP**)
  - at return types (**BIMOR**)

We next explain each of them in turn, together with the sound alternative.

3.1 Unsoundness Caused by Optional Typing

We distinguish between four sources of unsoundness concerning the optional typing mechanism (and hence the type `dynamic`). In comparison, Ernst et al. [9] simply forbid `dynamic` altogether, which is unnecessarily strict.

***Type `dynamic` at lookups* (DLOOK)** Quoting the Dart language specification [7, Section 19.6]: "Type `dynamic` has methods for every possible identifier and arity, with every possible combination of named parameters. These methods all have `dynamic` as their return type, and their formal parameters all have type `dynamic`." Fields are accessed via getters and setters, which behave similarly. It should also be noted

[2] MNU and RSV correspond to TypeError in Siek et al. [28], and SV corresponds to CastError; moreover, Siek et al. express the criteria slightly differently via a blame tracking mechanism, which Dart does not have.

[3] One could choose to also treat RSV errors as forbidden, in which case RSV should be included here alongside MNU and SV errors.

that function calls technically involve lookups of a special `call` property, so attempts to call non-function values may result in MNU errors.

The following program, where `x` is declared without a type annotation and therefore has static type `dynamic`, is well typed.

```
3 var x = 5;
4 x.method();
```

This is clearly unsound as an MNU error appears at line 4.

A natural way to fix this kind of unsoundness is to change the static type system to raise a warning on all accesses on type `dynamic` for which the accessed method or field is not declared in the `Object` class. (Note that we are not actually suggesting to make this change; we merely point out what a sound alternative would look like.)

***Type `dynamic` with ground types* (DGND)** Another key property of type `dynamic` is that it is a subtype of any type, so the type system permits, for example, assignments of the form $x = e$ when expression e has static type `dynamic` irrespective of the declared type of the variable x. For the DGND category, we consider only the cases where the declared type of x is a ground type (i.e., not a generic type, a type parameter, or a function type); other cases are covered later by DGEN and DFUN. Such assignments can obviously break soundness, as shown by the following example.

```
5 var x = 5;
6 num y = x;
7 String z = x;
8 print(z + z);
9 z.substring(0, 5);
```

The variable `x` has static type `dynamic`. The assignments in line 6 and 7 and the invocation in line 9 are all allowed by the type checker. At runtime, however, there is no `dynamic` type involved: the runtime type of `x` is `int`. In checked mode, subtype checks are performed at lines 6 and 7. The first runs fine, because `int` $<:$ `num`, but an SV error occurs at line 7. In production mode, no subtype checks are performed, but an RSV error occurs at line 8 and an MNU error at line 9. Note that the operator `+` is present in both the `int` and `String` classes (operators are methods in Dart), so line 8 gives no error and 10 is printed.

If all sources of unsoundness except DGND are fixed, then there can be no untrapped errors in checked-mode execution of well-typed programs, thereby fulfilling criterion 2 for gradually typed soundness (see Section 2).

Type `dynamic` is assignable to any type T for *two* reasons: first, because `dynamic` is a subtype of T, and second, because assignability is symmetric and `dynamic` is also a supertype of T. This means that fixing DGND unsoundness requires two modifications: we simply disallow those two cases when T is a ground type. However, to reduce overlap with DFUN this change does not apply when assignability is used for checking subtyping of function types.

***Type `dynamic` with generic types* (DGEN)** The type `dynamic` can be used as type argument in generic classes, in which case it is reified at runtime. As an example, `List<dynamic>` can appear both as a static type and as a dynamic type, unlike `dynamic`, which can only be a static type. The DGEN category of unsoundness pertains to the assignability relation allowing assignments of the form $x = e$ where the type of x is generic and the static type of e either is `dynamic` or is generic and contains `dynamic` at a position where the counterpart in the type of x is non-`dynamic`. An example is that x has type `List<List<int>>` and e has type `dynamic`, `List<dynamic>`, or `List<List<dynamic>>`, which are all allowed by the type checker. The following program demonstrates why DGEN may lead to unsoundness.

```
10 List<dynamic> y = new List<dynamic>();
11 y.add(true);
12 List<int> x = y;
13 x.first.toDouble();
```

The program is well typed but fails with an MNU error at line 13 because `toDouble` is not defined for booleans. The subtype check at line 12 in checked mode execution does not catch the error, because `y`'s runtime type `List<dynamic>` is a subtype of `List<int>`.

This example demonstrates an RSV error: reading `x.first` in line 13 yields a boolean, which does not match the declared type `int`.

Notice that the DGEN category of unsoundness is different from DGND. The subtype check performed at line 12 (in checked mode) is insufficient for catching the error, making it untrapped and unnoticed until the lookup of `toDouble` in line 13 where it causes the MNU error. In contrast, DGND can only result in trapped errors. The DGEN category therefore violates criterion 2 for gradually typed soundness.

Our fix for this kind of unsoundness is similar to DGND, except that we now consider the cases where T is generic rather than ground.

***Type `dynamic` with function types* (DFUN)** As for generic types, function types can contain `dynamic`. The DFUN category is exactly like DGEN except using function types instead of generic types. For example, DFUN comprises assignments $x = e$ where x has type `int`→`int` and e has type `dynamic` or `int`→`dynamic`. (We here ignore the parts in negative position in the function types, so DFUN does not include the case where e has type `dynamic`→`int`, for example.) Those cases are allowed by the type checker but may cause unsoundness, as the following example shows.

```
14 typedef int intmap(int x);
15 dynamic intmapimpl(int x) {
16   return true;
17 }
18 intmap x = intmapimpl;
19 x(5).toDouble();
```

The program is well typed but fails with an MNU error at line 19 in both production mode and checked mode. The

static type of `intmapimpl` in line 18 is `int→dynamic`, and `x` has type `int→int`. Similar to the DGEN example, the checked-mode runtime type checks are not strong enough to trap this mismatch early at line 18.

To fix the DFUN category of unsoundness, we modify subtyping and assignability in the same way as for DGND and DGEN, except this time for function subtyping.

3.2 Other Sources of Unsoundness

There are other sources of unsoundness of Dart's type system that are not concerned with the optional typing mechanism (and thereby violate criterion 1 for gradually typed soundness, cf. Section 2).

Symmetric assignability **(SYA)** Dart's type system allows an expression of type T to be assigned to a variable, parameter, or field of type S whenever $T <:> S$. In checked-mode execution, an SV error occurs if not $T <: S$.

The following program shows why this is unsound.

```
20 int x = new Object();
21 x + 5;
```

The type checker raises no warnings. In checked mode, an SV error appears at line 20. In production mode, reading `x` in line 21 gives an RSV error, which is untrapped, so execution proceeds until the lookup of `+`, resulting in an MNU error.

The natural fix to SYA unsoundness is unsurprising: simply restrict assignability to allow only the case where $T <: S$ but not the converse.

Covariant generics **(COGE)** Dart's subtyping for generics is unsound, because it allows covariance (similar to subtyping for arrays in Java): we have `T<`E`>` $<:$ `T<`K`>` if $E <: K$ (technically, this is using a *type specificity* relation rather than subtyping). Consider the following class hierarchy where `B` $<:$ `A` due to the inheritance.

```
22 class A { }
23 class B extends A {
24   void m() {}
25 }
```

The following code is well typed.

```
26 List<A> x = new List<B>();
27 x.add(new A());
```

The assignment in line 26 is allowed statically because of the covariant generics rule. In checked mode execution an SV error appears at line 27. Notice that with checked mode, COGE unsoundness cannot introduce untrapped errors (i.e., RSV errors), in contrast to DGEN and DFUN unsoundness.

The traditional sound choice for generics, if variance annotations are not used, is to require invariance, that is, `T<`E`>` $<:$ `T<`K`>` if $E = K$ [9]. To reduce the overlap with DGEN we choose a slightly more fine-grained alternative that also allows $E =$ `dynamic`. In this way, our sound choice for COGE will reject `List<A> x = new List<B>()` but allow `List<A> x = new List<dynamic>()`.

Function subtyping **(BIFSP and BIFSR)** Dart's type rule for function subtyping uses bivariance, both for the parameter types and for the return types: for functions with one parameter, a function type $T \to S$ is a subtype of another one $T' \to S'$ if $T <:> T'$ and $S <:> S'$ (or $S =$ `void`).

Both uses of bivariance result in unsoundness. We denote these sources of unsoundness by BIFSP and BIFSR, respectively. In the following example, the declared type of `x` and the runtime type of `funimpl` are `Object` $\to$ `int` and `int` $\to$ `Object`, respectively.

```
28 typedef int fun(Object x);
29 Object funimpl(int x) {
30   return x.isOdd;
31 }
32 fun x = funimpl;
```

The program is well-typed, and the assignment in line 32 passes the subtype check in checked-mode execution. Because of BIFSR, the following invocation is well typed, but it encounters an RSV error since the call returns a boolean while an `int` is expected according to the declared type, and shortly after it fails with an MNU error since `toDouble` is not declared for booleans.

```
33 x(5).toDouble();
```

Likewise, because of BIFSP the following invocation is well typed, but in production mode it fails with an MNU error since `isOdd` is not declared for booleans, and in checked mode it fails with an SV error since `true` is not an integer.

```
34 x(true);
```

The natural way to fix these kinds of unsoundness is to require contravariant parameter types (for BIFSP) and covariant return types (for BIFSR) in function subtyping.

Method overriding **(BIMOP and BIMOR)** The type rule for overriding of methods in subclasses uses bivariance, in the same way as function subtyping: if a method `m` has type $T \to S$ in a class `A` and `m` is overridden with type $T' \to S'$ in a subclass `B` of `A`, it is required that $T <:> T'$ and $S <:> S'$. Since fields are implicitly accessed as getters and setters, this applies to fields too.

In parallel with function subtyping, these two uses of bivariance also result in unsoundness, which we denote BIMOP and BIMOR, respectively. Examples corresponding to lines 28–34 can therefore be made using method overriding:

```
35 class A {
36   int m(Object x) => 0;
37 }
38 class B extends A {
39   Object m(int x) => x.isOdd;
40 }
41 A s = new B();
42 s.m(5).toDouble();
43 s.m(true);
```

Category	MNU	SV	RSV	<:	Type Checker	Runtime
DLOOK	•	-	-	-	54	-
DGND	-	•	-	(⋆)	6	-
DGEN	•	•	•	⋆	25	32
DFUN	•	•	•	⋆	28	13
SYA	-	•	-	-	3	-
COGE	-	•	-	⋆	10	8
BIFSP	-	•	-	⋆	6	1
BIFSR	•	•	•	⋆	8	2
BIMOP	-	•	-	-	3	-
BIMOR	•	•	•	-	3	-
total LOC					121	42

Figure 1: Sources of unsoundness. A • indicates that the error may occur in checked-mode execution of a well-typed program if the given source of unsoundness is not fixed. A ⋆ indicates that subtyping is affected (and thereby also the meaning of SV and RSV) by the suggested fix for that source of unsoundness. The columns 'Type Checker' and 'Runtime' show the LOC in our implementation of the fix. The 'total LOC' row shows the LOC involving all the fixes (some of them overlap).

Line 42 demonstrates BIMOR, similar to line 33 for BIFSR, and line 43 demonstrates BIMOP (assuming the preceding line is removed), similar to line 34 for BIFSP.

Note that Ernst et al. [9] have chosen to express the type rule for method overriding via the definition of function subtyping; instead, we treat them separately to allow experimenting with making function subtyping sound without necessarily also making method overriding sound.

To fix these kinds of unsoundness, we accordingly restrict the typing for method overriding to contravariant parameter types (for BIMOP) and covariant return types (for BIMOR).

3.3 Discussion

Figure 1 summarizes the sources of unsoundness and shows for each of them which errors may occur in checked-mode execution of a well-typed program if the unsoundness is not fixed. As RSV errors are untrapped, possibility of RSV implies possibility of MNU and SV. The figure also shows which fixes affect the subtyping relation (and thereby also the meaning of SV and RSV and the runtime type checks performed in checked-mode execution[4]), and the lines of code in our implementation of the fixes in the type checker and runtime system, respectively.

With this foundation in place, we can explore different soundness configurations, by selecting for which categories of unsoundness to apply the suggested fix. It follows from Figure 1 that some configurations have particularly interesting properties:

full type safety If we apply the suggested fix to all 10 categories, then the type system becomes sound (in the sense defined in Section 2).[5]

message safety We obtain the notion of message safety [9], meaning that well-typed programs cannot have MNU errors, if we apply the fixes for DLOOK, DGEN, DFUN, BIFSR, and BIMOR.

gradual safety We can satisfy the two criteria for gradually typed soundness from Section 2 by applying the fixes for all categories of unsoundness, except DGND. The resulting type system will hence only allow optional typing for ground types.

RSV safety Absence of RSV errors can be guaranteed by the type checker if we apply the fixes for (only) DGEN, DFUN, BIFSR, and BIMOR. This is interesting because RSV errors are untrapped, even in checked mode, and can indirectly cause MNU and SV errors.

In the following section, we investigate experimentally to what extent the different configurations may affect the type warnings and errors for existing Dart programs.

4. Experiments

To test our hypothesis we have downloaded the latest version of every library and application hosted by the Dart 'pub' repository,[6] consisting of 2 093 projects. Unfortunately, we have to exclude some projects, which are outdated or incompatible with the Dart version we use (v1.16.1), or for which we are unable to automatically identify the files to analyze. Our experiments are therefore performed on a total of 1 883 projects, consisting of 2.4 M LOC.

For each category of unsoundness, we can choose in our implementation whether or not to enable the sound alternative, leading to 2^{10} possible configurations of the type system. Each configuration is characterized by a set of unsoundness categories being fixed, for example, $\emptyset$ corresponds to the standard Dart type system, and $\{$COGE, SYA$\}$ is the type system where we have applied the suggested sound alternatives for covariant generics and symmetric assignability. The *size* of the configuration is the number of unsoundness categories in the set.

As shown in the two rightmost columns in Figure 1, implementing the sound alternatives for all categories of unsoundness can be accomplished with only 121 LOC in the type checker and 42 LOC in the runtime system. Naturally, more expressive type systems, for example with support for generic methods, wildcards, or variance annotations will be substantially more complex, but these numbers show that

[4] Fixing DGND does affect subtyping, but not runtime execution since no object has type `dynamic`.

[5] This is less restrictive than the requirements in Ernst et al. [9], which forbid all uses of `dynamic`. As an example, the following code is well typed in our sound type system, even though it uses `dynamic`: `dynamic x = "string"; (x as String).substring(0,3)`

[6] `https://pub.dartlang.org/` (May 2nd 2016)

unsoundness by itself does not save any significant effort for the language implementors.

We divide the hypothesis into three parts. First, we test for each category of unsoundness whether or not switching to the sound alternative results in a significant increase in the number of warnings raised by the static type checker. Second, we perform a preliminary qualitative study of some of the new warnings. Third, we test how the choice of configuration affects the runtime errors in checked-mode execution of the test suites.

Due to the limited space we report only highlights of the experiments. Our entire dataset, including details about warnings and errors in different type system configurations, is available online.[7]

4.1 Static Type Checker Warnings

We perform the following experiment. We run the static type checker for different configurations. We first measure the number of additional warnings per KLOC in each benchmark, using the $\emptyset$ configuration as baseline. Figure 2 shows the resulting distributions for all configurations of size 1 and for RSV safety $\text{RSV} = \{\text{DGEN}, \text{DFUN}, \text{BIFSR}, \text{BIMOR}\}$ and "bivariance" $\text{BI} = \{\text{BIFSP}, \text{BIFSR}, \text{BIMOP}, \text{BIMOR}\}$; we later report on the results for configurations that involve other combinations of categories. We also measure for each configuration, for how many benchmarks we observe no more than 5 new warnings, again using $\emptyset$ as baseline. (For a typical project, 5 or fewer new warnings can hardly be called a "significant increase", cf. the hypothesis.) These numbers are also shown in Figure 2, below each column.

Unsurprisingly, restricting the optional typing mechanism without adding any form of type inference gives a high number of warnings in all the benchmarks, as shown by Figure 2 for $\{\text{DLOOK}\}$, $\{\text{DGND}\}$, and $\{\text{DGEN}\}$. It is even a recommended programming style to omit type annotations at local variables,[8] which obviously results in many warnings, especially for DLOOK and DGND. For someone with little experience with Dart, probably the most surprising numbers are those for $\{\text{DGEN}\}$. More than 50% of the benchmarks contain more than 21 (and in some cases beyond 100) occurrences of DGEN per KLOC. Section 4.2 provides a possible explanation. The numbers for $\{\text{DFUN}\}$ indicate that type `dynamic` is not used as much in combination with function types, compared to the other sources of unsoundness involving `dynamic`.

The numbers for $\{\text{BIFSP}\}$, $\{\text{BIFSR}\}$, $\{\text{BIMOP}\}$, and $\{\text{BIMOR}\}$ are more interesting. We see that switching to the sound alternative for each of those categories results in almost no new warnings, if ignoring a few outlier benchmarks. We get only 0.3 new warnings/KLOC, and in only 1% of the benchmarks we get more than 5 new warnings. This result seems to contradict the hypothesis, if only static warnings are considered: *each of these sources of unsoundness can be fixed without significantly affecting the number of warnings issued by the type checker*.

The results for $\{\text{SYA}\}$ and $\{\text{COGE}\}$ are not conclusive. These configurations give 2.4 and 5.5 warnings/KLOC in average and significantly affect 9% and 18% of the benchmarks, respectively.

Our observations described so far are for size 1 configurations, however, there are cases where combinations of fixes may lead to warnings that are not raised by the fixes in isolation. In other words, there can be an interplay between the different sources of unsoundness, as in this simple assignment: `List<dynamic> x = new List<int>()`. This is well typed with any of the configurations $\emptyset$, $\{\text{SYA}\}$, or $\{\text{COGE}\}$, but a warning appears with $\{\text{SYA}, \text{COGE}\}$. Our fix to COGE unsoundness prohibits `List<int>` $<:$ `List<dynamic>` but still allows the converse, so the assignment is allowed unless SYA unsoundness is also fixed. To investigate whether such interplay affects our conclusions, we compare the set of warnings we get from the BI configuration with the union of the warnings we get from each of the four singleton configurations. The result is that only 1% of the warnings for BI can be attributed to such effects. Since each of BIFSP, BIFSR, BIMOP, and BIMOR can be fixed individually without leading to a significant increase in warnings, it is interesting to notice that the combination BI also has this property: *the* BI *configuration results in more than* 5 *new warnings compared to the standard Dart type system at only* 2.8% *of the benchmarks*.

Similarly, the RSV configuration is interesting for the reason explained in Section 3.3. However, due to the large number of warnings that result from our fix for DGEN unsoundness, RSV affects many benchmarks significantly.

In summary, we see that for some sources of unsoundness, switching to a sound alternative can be done without significantly affecting the number of type warnings in many benchmarks, which means that such a modification would not cause programmers to become overwhelmed with annoying warnings from the type checker. Still, two questions remain. First, in type system configurations that do result in a significant increase in warnings, it may be the case that the warnings are "good" in the sense that they indicate issues in the code the programmer likely would want to fix (we study this in Section 4.2). Second, it is possible that type system modifications that affect subtyping cause a low number of warnings but still result in runtime errors that did not occur with the original type system (see Section 4.3).

4.2 Qualitative Study of Type Warnings

As motivated above, it is relevant for our hypothesis to test whether warnings that appear in one of the modified type systems but not in the original one are typically "good" or "bad" in the following sense.

[7] `http://www.brics.dk/undart/`

[8] `https://www.dartlang.org/effective-dart/usage/`

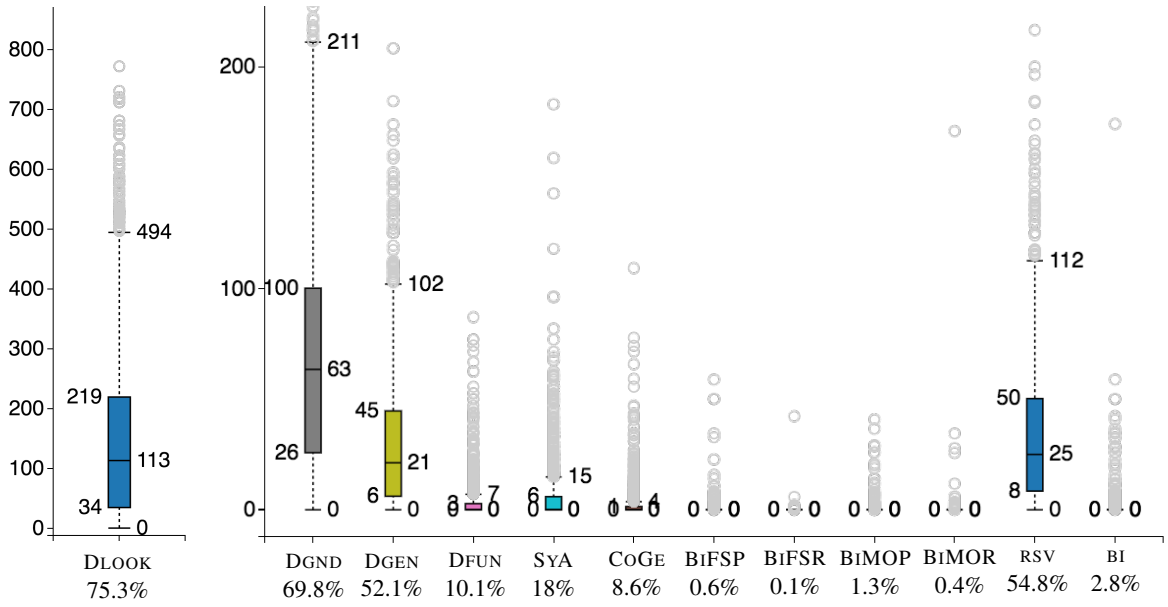

Figure 2: Type checker warnings for selected configurations. Below each label we report the percentage of benchmarks affected by that configuration. The plot represents the distribution of warnings/KLOC of each of them. The top- and lower-most ticks of each bar represent the maximum and minimum number of warnings per KLOC of the entire dataset, outliers excluded. The space between the two ticks is divided into four parts by three bars, called 1st, 2nd and 3rd *quartile*. The colored box, delimited by the 1st and 3rd quartile, shows where 50% of warnings/KLOC lie. The 2nd quartile is the median of the dataset. The outliers are represented by circles. For typographical reasons, we cap a few outliers, and we draw two y axes with different scale: one for DLOOK and one for the other configurations.

- A *good* warning is one that indicates a programmer mistake that is easy to fix by only changing a few type annotations. (Note that such warnings do not necessarily indicate serious bugs, but possibly minor mistakes that only affect code maintainability.)
- A *bad* warning is an artifact from the type system, that is, a warning that cannot be fixed easily, but would require either a nontrivial modification of the program code, a more expressive type system, or changes in the standard library.

To investigate this, we have performed a preliminary manual study of 51 randomly selected warnings that do not appear with the $\emptyset$ configuration, and categorized each warning into one of the two kinds.[9] Clearly, our classification is somewhat vague and inevitably subjective, but we strive to be conservative and only categorize a given warning as "good" if we consider that programmers would likely fix the mistake but just have never been warned about it due to the unsoundness of the standard type system.

In order to study the correlation between specific sources of unsoundness and the ratio between good and bad warnings, we automatically categorize each warning by the minimal set of unsoundness fixes that are necessary for the warning to be exposed. We call such a minimal set an *origin* of the warning. Note that a warning can have multiple origins.

Table 1 summarizes the results. The listed 11 origins account for all the 51 warnings. For example, we find that all 9 warnings examined from the {BIMOR} group are due to easily fixable oversights by the programmer. A typical example is the following field that is declared with type `num`, which unsoundly overrides the type `int` in the super-class:

```
44 class Displ implements Display {
45   num get viewOffset;
46   ...
47 }
```

Changing the type annotation from `num` to `int` fixes the problem (and does not introduce any new warnings).

Overall, roughly $2/3$ of the warnings are categorized as "good". Although the numbers in this preliminary qualitative study are admittedly low, they do indicate an interesting trend: the majority of the warnings indicate issues in the code that are easy to fix by changing a few type annotations. The primary benefit of making those changes is that it strengthens the role of type annotations as documentation to the programmers. "Unsound" type annotations are less informative and can be misleading to programmers reading the code.

Interestingly, we find that *none* of the 20 warnings in the "bad" category are caused by programmers exploiting the

[9] More than 51 would obviously be better, but this kind of study is laborious. Arguing that warnings are "good" requires us to make sure that corrected annotations do indeed work, which is nontrivial especially for library code.

Origin	{BIFSP}	{BIFSR}	{BIMOP}	{BIMOR}	{COGE}	{SYA}	{DGEN}	{COGE, DGND}	{COGE, BIMOP}	{COGE, SYA}
good	5	1	8	9	2	1	1	0	1	1
bad	6	3	3	0	4	2	1	1	0	0

Table 1: Classification of warnings.

flexibility provided by unsoundness to do clever things: most of them are either consequences of how the standard library has been designed or of the lack of generic methods.

For example, list literals have type `List<dynamic>`, so the assignment `List<int> x = [1, 2, 3]` results in a warning with the configuration {DGEN}. Such warnings could clearly be avoided with a slightly less naive language design, without sacrificing soundness. The signature of the `List.map` method is `Iterable<dynamic> map(dynamic f(E e))` due to the lack of generic methods. This has the unfortunate consequence that `dynamic` appears frequently as type parameter, making it practically impossible to write Dart programs that do not use `dynamic`. We also see examples where generic methods could avoid warnings related to BIFSR.

Another interesting type system artifact is that {COGE} conflicts with asynchronous functions, as in the following example.

```
48 Future<String> generate(Element element) async {
49     if (element is! LibraryElement)
50       return null;
51     ...
52 }
```

Asynchronous expressions and the type `Future<`T`>` are treated specially by Dart's type system. If an expression of type T is returned from an asynchronous function, the type system ascribes `Future<`T`>` as the type returned by the function body. In the example above, `Future<`$\bot$`>` requires COGE unsoundness to match the `generate` function annotated return type, which is `Future<String>`, leading to a warning at line 50 for the configuration {COGE}. However, since type `Future<`T`>` already has a special meaning in the type system, and futures with value null also have a special meaning, it would be quite natural to treat this combination specially too, without necessarily resorting to unsoundness.

The HTML API in the standard library also encourages the use of symmetric assignability. For example, it is common to write `DivElement d = new Element.tag("div")` knowing that the constructor returns an object of type `DivElement` although its static return type is the super-type `Element`. Note that in this case the programmer could instead create the object by `new DivElement()` (or use an explicit type cast) and thereby not rely on SYA unsoundness.

In summary, we find that a majority of the warnings in this small qualitative study are "good", and that the "bad" ones are not due to programmers exploiting unsoundness in clever ways but rather due to the design of the standard library and the lack of generic methods.

4.3 Runtime Errors

Since some of our type system modifications affect subtyping, which is used at runtime, it is possible in some configurations that well-typed programs encounter more runtime errors in the benchmark code than with the standard type system. To investigate the extent of this situation, we have modified the Dart Virtual Machine (VM) for the five relevant sources of unsoundness according to Figure 1: DGEN, DFUN, COGE, BIFSP, and BIFSR.

We have then collected all the functioning test files from our benchmarks, i.e. the ones that run without any custom setup and that pass with the standard Dart VM. (A test file is the atomic test unit that 'pub' can handle, although a single file may contains multiple test procedures.) This gives us a total of 1 032 succeeding test files among 390 of the benchmark projects.

Table 2 shows the number of tests that are affected by the modified subtyping relation in various configurations. Most importantly, we see that only 1.8% of the tests are affected by the BI configuration (which is equivalent to {BIFSP, BIFSR} regarding runtime behavior). Manually inspecting the affected tests, however, reveals that all of them are artifacts caused by implementation choices in the VM, for example, where field declarations are internally desugared in a way that introduces function assignments, causing the VM to perform needless subtype checks. Since the number of affected tests is low, and the number could even be reduced further by different VM implementation choices, this result corroborates and completes the conclusions from the previous sections.

For DGEN, DFUN, and COGE, the numbers of affected tests are much higher. The number for DGEN reinforces the observation in Section 4.2 that with the current design of lists, maps, and sets in the standard library, most such objects inevitably hold a `dynamic` type argument. This also shows that in many assignments of the kind `C<T> x = y`, the type of `y` is `C<dynamic>`, which means that the `T` annotation is completely superfluous from a type safety perspective.

Configuration	Affected
{DGEN}	67.3% (69.2%)
{DFUN}	50.9% (50.1%)
{COGE}	81.3% (77.7%)
{BIFSP}	1.6% (3.3%)
{BIFSR}	0.7% (1.8%)
{BIFSP, BIFSR}	1.8% (4.1%)

Table 2: Percentage of tests affected in different configurations (projects affected are shown in parentheses).

5. Related Work

Data-driven language design Several researchers have in recent years pointed out the need for supplying evidence to support language design claims. Stefik and Hanenberg [30] "seriously question the scientific validity of any claim made by a language designer that does not have actual data in hand." According to Markstrum [18], "We should be more aware of valiant and effective efforts for supplying evidence to support language design claims." Murphy-Hill and Grossman [22] argue that controlled experiments and field studies in usage of language features rarely have influenced language design, but they predict: "Over the next decade, language designers will increasingly use data to drive the design of new languages and language features." Our work takes a small step in this direction by providing a methodology and results about type system unsoundness in Dart.

Empirical evaluation of type systems Empirical evaluation studies of programming language design is typically based on controlled experiments or field studies aiming to measure programmer productivity, or code repository analysis to measure prevalence of language features. We here focus on the literature that is concerned with type systems. An early example is the evaluation of defect-detection capabilities of inter-module type checking in C by Prechelt and Tichy [24].

Souza and Figueiredo [29] have studied Groovy projects to investigate how programmers use optional typing, and Eshkevari et al. [10] have evaluated the design of Hack's type system by analyzing the types that occur in PHP programs.

The use of generic types in Java has been studied by Parnin et al. [23]. Similar to our study, this was made after the language design was decided, rather than guiding the design, and it too was based on open source programs. Conversely, their focus was on measuring adaption of a new language feature, whereas our study involves a feature that has been there since the first version of the language. Another question is whether or not generic types increase programmer productivity [15].

Several controlled experiments have been conducted to study the potential benefits of static typing, most notably by Hanenberg et al. [8, 13, 14]. We are aware of only one such study involving Dart: Faldborg and Nielsen [12] report on a small user study that investigates the effect of using statically typed APIs.

None of these studies focus on type system unsoundness.

Gradual typing and other languages Dart is sometimes called gradually typed in the sense that it supports a gradual evolution from untyped to typed code. As discussed in Section 2, the term "gradual typing" is often associated with soundness guarantees, which Dart obviously violate. However, in our view it is mostly the lack of contracts and blame tracking that makes Dart fundamentally different from most gradually typed languages. This design, combined with the use of nominal subtyping, makes the checked-mode runtime type checks relatively fast, unlike other languages with optional typing [25, 31, 34]. It would be interesting to evaluate empirically whether the blame tracking in other languages in practice has advantages that justify its cost in performance.

The Dart strong mode initiative aims to provide a sound alternative to Dart's standard type system [20]. This goes considerably further than the minor adjustments we have explored in this paper, for example, also adding local type inference and generic methods. It is unclear whether strong mode will eventually be part of the language standard. Still, our approach and results may be useful for qualifying the design decisions.

TypeScript is an extension of JavaScript, with an optional type system that is unsound by design, in many ways like Dart. The sources of unsoundness in TypeScript have been characterized [1], and the connections to gradual typing have been established [25]. To our knowledge no empirical studies have been made to justify the design choices, but our approach could in principle be applied to this language too.

Unsoundness in static analysis Deliberate unsoundness is rare in type systems, but less so in static analyzers [16]. Recently, Christakis et al. [6] have studied how unsound assumptions in a static analyzer affect error detection capabilities. As in our work, that study involved modifying the analysis tool, experiments on open source projects, and use of test suites to locate the sources and measure the consequences of unsoundness.

6. Conclusion

We have presented the first empirical study of how unsoundness in a type system is being used in real programs. Considering the hypothesis stated in Section 1, based on our experiments we conclude that some but not all sources of unsoundness in Dart's type system can be justified empirically.

Unless one is willing to make nontrivial extensions to the type system (e.g. generic methods and local type inference) it is difficult to obtain full type safety, message safety, or even gradual safety (as defined in Section 3.3) without causing a significant increase in the number of static type warnings or runtime type errors in existing Dart code.

However, we find that, especially, bivariant function subtyping (BIFSP and BIFSR) and method overriding (BIMOP and BIMOR) could easily be replaced by sound alternatives without overwhelming the programmers with annoying type warnings or runtime errors. Moreover, our preliminary qualitative study of type warnings suggests that programmers rarely exploit the flexibility provided by unsoundness (but the standard library does take advantage of it), and that eliminating some of the sources of unsoundness would likely result in more "good" warnings than "bad" warnings. Finally, our implementation of the sound alternatives shows that unsoundness does not necessarily save any significant effort for the language implementors.

These results demonstrate that it may be worthwhile to explore further how alternatives to Dart's current type system may affect programmer productivity, for example, via controlled experiments. Another opportunity for future work is to focus on symmetric assignability and covariant generics, for which our results are still inconclusive. It would also be interesting to investigate alternative designs of the standard library to reduce its dependency on type system unsoundness.

Acknowledgments We thank Erik Ernst and the reviewers for the insightful comments and Vyacheslav Egorov for his support on the Dart VM internals. This work was supported by the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation program (grant agreement No 647544).

References

[1] G. M. Bierman, M. Abadi, and M. Torgersen. Understanding TypeScript. In *Proc. European Conf. on Object-Oriented Programming (ECOOP)*, 2014.

[2] G. Bracha. Pluggable type systems. In *OOPSLA Workshop on Revival of Dynamic Languages*, 2004.

[3] G. Bracha. *The Dart Programming Language*. Addison-Wesley, 2015.

[4] E. Brandt. Why Dart types are optional and unsound, 2011. `https://www.dartlang.org/articles/why-dart-types/`.

[5] L. Cardelli. Type systems. In *The Computer Science and Engineering Handbook*, pages 2208–2236. CRC Press, 1997.

[6] M. Christakis, P. Müller, and V. Wüstholz. An experimental evaluation of deliberate unsoundness in a static program analyzer. In *Proc. 16th Int. Conf. on Verification, Model Checking, and Abstract Interpretation (VMCAI)*, 2015.

[7] Ecma International. *Dart Programming Language Specification, ECMA-408, 4th Edition*, December 2015.

[8] S. Endrikat, S. Hanenberg, R. Robbes, and A. Stefik. How do API documentation and static typing affect API usability? In *Proc. 36th Int. Conf. on Software Engineering (ICSE)*, 2014.

[9] E. Ernst, A. Møller, M. Schwarz, and F. Strocco. Message safety in Dart. *Science of Computer Programming*, 2016. In press. Earlier version in Proc. 11th Symp. on Dynamic Languages (DLS), 2015.

[10] L. M. Eshkevari, F. D. Santos, J. R. Cordy, and G. Antoniol. Are PHP applications ready for Hack? In *Proc. 22nd IEEE Int. Conf. on Software Analysis, Evolution, and Reengineering (SANER)*, 2015.

[11] Facebook. Hack, May 2016. `http://hacklang.org/`.

[12] M. Faldborg and T. L. Nielsen. Type systems and programmers: A look at optional typing in Dart. Master's thesis, Aalborg University, 2015.

[13] S. Hanenberg. An experiment about static and dynamic type systems: doubts about the positive impact of static type systems on development time. In *Proc. 25th Annual ACM SIGPLAN Conf. on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2010.

[14] S. Hanenberg, S. Kleinschmager, R. Robbes, É. Tanter, and A. Stefik. An empirical study on the impact of static typing on software maintainability. *Empirical Software Engineering*, 19(5):1335–1382, 2014.

[15] M. Hoppe and S. Hanenberg. Do developers benefit from generic types?: an empirical comparison of generic and raw types in java. In *Proc. ACM SIGPLAN Int. Conf. on Object Oriented Programming Systems Languages & Applications (OOPSLA)*, 2013.

[16] B. Livshits, M. Sridharan, Y. Smaragdakis, O. Lhoták, J. N. Amaral, B.-Y. E. Chang, S. Z. Guyer, U. P. Khedker, A. Møller, and D. Vardoulakis. In defense of soundiness: A manifesto. *Commun. ACM*, 58 (2):44–46, 2015.

[17] A. M. Maidl, F. Mascarenhas, and R. Ierusalimschy. A formalization of typed Lua. In *Proc. 11th ACM Symp. on Dynamic Languages (DLS)*, 2015.

[18] S. Markstrum. Staking claims: A history of programming language design claims and evidence. In *Workshop on Evaluation and Usability of Programming Languages and Tools (PLATEAU)*, 2010.

[19] E. Meijer and P. Drayton. Static typing where possible, dynamic typing when needed: The end of the cold war between programming languages. In *OOPSLA Workshop on Revival of Dynamic Languages*, 2004.

[20] V. Menon et al. Strong mode, March 2016. `https://github.com/dart-lang/dev_compiler/blob/master/STRONG_MODE.md`.

[21] Microsoft. TypeScript language specification, February 2015. `http://www.typescriptlang.org/Content/TypeScript%20Language%20Specification.pdf`.

[22] E. R. Murphy-Hill and D. Grossman. How programming languages will co-evolve with software engineering: a bright decade ahead. In *Proc. Future of Software Engineering (FOSE)*, 2014.

[23] C. Parnin, C. Bird, and E. R. Murphy-Hill. Adoption and use of Java generics. *Empirical Software Engineering*, 18(6):1047–1089, 2013.

[24] L. Prechelt and W. F. Tichy. A controlled experiment to assess the benefits of procedure argument type checking. *IEEE Trans. Software Eng.*, 24(4):302–312, 1998.

[25] A. Rastogi, N. Swamy, C. Fournet, G. M. Bierman, and P. Vekris. Safe & efficient gradual typing for TypeScript. In *Proc. 42nd ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages (POPL)*, 2015.

[26] A. Rastogi, N. Swamy, C. Fournet, G. M. Bierman, and P. Vekris. Safe & efficient gradual typing for TypeScript. In *Proc. 42nd ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages (POPL)*, 2015.

[27] J. G. Siek and W. Taha. Gradual typing for functional languages. In *Scheme and Functional Programming Workshop*, 2006.

[28] J. G. Siek, M. M. Vitousek, M. Cimini, and J. T. Boyland. Refined criteria for gradual typing. In *Proc. 1st Summit on Advances in Programming Languages (SNAPL)*, 2015.

[29] C. Souza and E. Figueiredo. How do programmers use optional typing?: an empirical study. In *Proc. 13th Int. Conf. on Modularity (MODULARITY)*, 2014.

[30] A. Stefik and S. Hanenberg. The programming language wars: Questions and responsibilities for the programming language community. In *Proc. ACM Int. Symp. on New Ideas, New Paradigms, and Reflections on Programming & Software (Onward!)*, 2014.

[31] A. Takikawa, D. Feltey, B. Greenman, M. S. New, J. Vitek, and M. Felleisen. Is sound gradual typing dead? In *Proc. 43rd Annual ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages (POPL)*, 2016.

[32] S. Tobin-Hochstadt and M. Felleisen. The design and implementation of Typed Scheme. In *Proc. 35th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages (POPL)*, 2008.

[33] L. Tratt. Dynamically typed languages. *Advances in Computers*, 77: 149–184, 2009.

[34] M. M. Vitousek, A. M. Kent, J. G. Siek, and J. Baker. Design and evaluation of gradual typing for Python. In *Proc. 10th ACM Symp. on Dynamic Languages (DLS)*, 2014.