Type Safety Analysis for Dart

Thomas S. Heinze Anders Møller Fabio Strocco
Aarhus University, Denmark
{t.heinze,amoeller,fstrocco}@cs.au.dk

Abstract

Optional typing is traditionally viewed as a compromise between static and dynamic type checking, where code without type annotations is not checked until runtime. We demonstrate that optional type annotations in Dart programs can be integrated into a flow analysis to provide static type safety guarantees both for annotated and non-annotated parts of the code. We explore two approaches: one that uses type annotations for filtering, and one that uses them as specifications. What makes this particularly challenging for Dart is that its type system is unsound even for fully annotated code. Experimental results show that the technique is remarkably effective, even without context sensitivity: 99.3% of all property lookup operations are reported type safe in a collection of benchmark programs.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program Verification

Keywords type systems; optional types; static analysis

1. Introduction

The Dart programming language supports optional type annotations, allowing programmers to select which parts of the programs are statically type checked [7]. For pragmatic reasons, type checking in Dart is unsound by design. This means that even in fully annotated programs where the static type checker reports no warnings, runtime type errors are still possible. The language designers argue that this leads to a simple and intuitive type system that provides flexibility to the programmers while still enabling useful IDE tool support [3].

Although some IDE features, such as, type warnings and code navigation, do not require soundness, this design choice precludes the use of sound refactorings and optimizations based on type annotations. Moreover, runtime type errors may be acceptable in web apps, but not in more safety critical code, which makes it interesting to explore alternative designs. Another concern is that dynamic type checks in languages with optional or gradual typing affect runtime performance, sometimes with disastrous consequences [26].

Two kinds of runtime type errors may occur in Dart programs: a *message-not-understood* error appears if the program attempts to access a field or method that does not exist, and a *subtype-violation* error appears (in checked mode execution) if a value does not match the declared type at a write operation. Type annotations are optional; the default type `dynamic` effectively disables subtype violation checks for the variable or field in question.

Previous work has formalized a core of Dart to clarify its type system and identify the precise causes of potential type errors [8]. That work also established two variations of the static type system: one for *message safety*, which statically rules out message-not-understood errors while still allowing subtype-violation errors, and *full type safety*, which statically prevents both kinds of type errors, akin to more traditional static type checking. One important limitation of that work is that it provides no guarantees for program code that uses type `dynamic`. With this foundation in place, we now investigate opportunities for statically checking type safety also in program code that is not fully type annotated.

Existing work on combining optional types and type inference does not solve the problems for languages like Dart. For example, the type inference algorithm by Siek and Vachharajani [22] is based on unification, which is insufficient for object-oriented languages with subtyping, and the technique for ActionScript by Rastogi et al. [20] does not account for generic classes, nor for the kinds of unsoundness that are present in Dart's type system. We show that it is possible to integrate optional types into a static type analysis, for a language with generic classes and where the type annotations cannot always be trusted. Moreover, we show that even a context-insensitive analysis can provide high precision.

The main contributions of this paper are as follows.

- We present a type safety analysis designed for sound type checking, that integrates optional types for Dart. The analysis is capable of statically checking absence of runtime type errors in Dart code that may not be fully annotated. The key challenge in the analysis design is how to incorporate type annotations in a sound manner. We explore two main approaches: one that uses type annotations for

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

ACM 978-1-4503-4445-6/16/10. . . $15.00
DOI: http://dx.doi.org/10.1145/10.1145/2989225.2989226

filtering the flow of types through the program, and one that uses the annotations as specifications allowing more *modular* reasoning. Furthermore, we investigate how analysis precision and time can be improved by optimistically assuming that type annotations can be trusted.

- We experimentally evaluate the approach on real Dart programs. Among the results is that even a context insensitive analysis is capable of statically checking safety of 99.3% of all property lookup operations, including those in unannotated parts of the code. We also report on the effectiveness of the different analysis modes and identify opportunities for improving precision further.

The purpose of our program analysis is not to automatically add type annotations to programs, but to infer the dataflow through non-annotated code and statically check for potential type errors. Using the terminology of Palsberg and Schwartzbach [19], this is a safety analysis, not a type inference. We believe such an analysis may be useful for programmers who wish to gain confidence in their code, in the spirit of gradual typing [21]. Our experimental results may also be useful to qualify the discussion of language design choices and to guide the development of new software tools supporting Dart programmers.

2. The Dart Language

The Dart language, being developed by Google and standardized by Ecma [7], features interesting design choices in gradual type systems. Dart is an object-oriented language with a syntax that resembles Java, and with support for first-class functions. Type annotations are optional, and warnings raised by the static type checker do not preclude execution. Dart programs are executed either in checked mode or in production mode. In checked mode, a runtime subtype check is performed every time a value is passed to an entity with a declared type; a subtype-violation runtime type error (technically, a `TypeError` exception) occurs if the check fails. In production mode, all type annotations are ignored. Another kind of runtime type error is message-not-understood (a `NoSuchMethodError` exception), which occurs when field or method lookup operations fail.

Unlike most gradual type systems, the one in Dart is designed to be unsound, meaning that both kinds of runtime type errors can occur even in fully annotated programs (the precise causes of unsoundness have been studied previously [8]). This means that Dart's standard type checking strictly does not satisfy the defining properties of gradual typing according Siek et al. [21, 22, 24], even in checked mode.

The Dart language specification explicitly encourages development of alternative static checkers. In this work we present one such checker that can soundly check for runtime type errors, even for programs that are not fully annotated.

3. Optional Types & Flow Analysis

Our goal is to provide a program analysis that can conservatively check whether a Dart program may encounter runtime type errors. As an extreme case, consider Dart programs without any type annotations. For such programs, an obvious choice is to design some form of flow analysis that tracks the possible runtime types of all variables and expressions, as done previously for dynamically typed languages, such as, Scheme [10], Self [1], or JavaScript [16].

The situation becomes more interesting when we also consider type annotations. In a Dart program, any local variable, method/function parameter, return value, and class field may or may not have a type annotation. Moreover, even in a fully annotated program that passes standard type checking, type errors may appear at runtime. This raises the question of how to incorporate the type annotations that are present, in a way that does not compromise soundness of our analysis. As an example, consider the following program.

```
1  f(x) {
2    return x + ", world!";
3  }
4  String g(String y) {
5    return f(y);
6  }
7  main() {
8    print(g("Hello"));
9  }
```

Example 1: Optional type annotations.

A pure flow-analysis based technique would ignore the `String` annotations on line 4 and infer that only string values are ever stored in `y`. Furthermore, the possible values of `x` can only come from `y`, so the application of `+` (operators are technically methods in Dart) cannot fail (i.e., it does not give a message-not-understood error). This results in a new string value that is eventually returned from `g`, so that the return operation is also guaranteed to succeed (i.e., without a subtype-violation error in checked mode execution).

Such an analysis can use type annotations for *filtering* the flow of types. Using the declared type of, for example, `y`, we can model the checked-mode subtype checks and thereby rule out types flowing into `y` that are not subtypes of `String` (for example type `int` if assuming another call site `g(42)`).

In contrast to such a whole-program flow analysis, an analysis that uses the type annotations as specifications would become more *modular*. For example, since `g` has type annotations of both its input and its output, it is possible to analyze `main` separately from `f` and `g`. Such modularity can have several benefits, even when analyzing complete programs: (1) Modular analysis tends to be more robust to changes in the program code and for programs that may not yet be ready for execution, since the effects of changes on the analysis results only propagate within the component boundaries. (2) Using type annotations as specifications may lead to more meaningful warning messages in situations where the programmer deliberately uses type annotations that are more

general than strictly necessary. (3) In principle, the modular approach can lead to better scalability of the analysis. As flow analysis involves computing transitive closures of dataflow in each component, modularly analyzing a large number of small components may bring analysis complexity from cubic to linear in practice. This modular analysis approach aligns well with a recommended programming style to use type annotations at interfaces of components.

Since Dart has both class inheritance and first-class functions, some form of control-flow analysis is necessary. To this end, we build on existing program analysis techniques from object-oriented and functional programming. Class hierarchy analysis [6] is often effective for object-oriented languages, but insufficient if type annotations are absent or if first-class functions are used extensively, in which case flow-based analysis [16] is more suitable.

4. Trusting Type Annotations

If type annotations could always be trusted, integrating the type annotations would be a straightforward modification of the flow analysis: when modeling *reads* from a type annotated variable, simply ignore the incoming dataflow that is assigned to that value and use the type annotation as a description of the possible types that may appear (i.e., as an implicit *assumption*); similarly, when modeling *writes* to a type annotated variable, emit a warning if the incoming flow does not match the type annotation (i.e., as an implicit *assertion*). Unfortunately, type annotations in Dart cannot always be trusted, which makes such reasoning unsound, as the following example shows.

```
10 class Cell<T> { T f; }
11 void main() {
12   Cell x = new Cell();
13   Cell<String> y = x;
14   x.f = 42;
15   var z = y.f.substring(2);
16   print(z);
17 }
```

Example 2: Dynamic type arguments.

This program defines a generic class containing a field whose type is a type parameter of the class. The `main` function creates an object that has runtime type `Cell<dynamic>` (the default type parameter is `dynamic`, so `Cell` has the same meaning as `Cell<dynamic>`, and Dart does not use type erasure). On line 14, an integer is stored in the field of the object, which is fine since the field type is `dynamic`. However, `y` also holds a reference to the object, and `y` has declared type `Cell<String>`, so looking at the declaration of `y` on line 13 and the use of `y` on line 15, one would expect that `y.f` yields a string. In fact, it yields an integer, so the invocation of `substring` fails with a message-not-understood runtime error. We note that the standard type system in Dart does not catch this error, nor does the modified type system by Ernst et al. [8] because of the use of `dynamic`. The essence of the problem in this program is that the type argument in the annotation `Cell<String>` cannot be trusted. There are different ways a program analysis or type system could reject the program statically. An obvious candidate is to disallow line 13 where an object of type `Cell<dynamic>` is assigned in a variable of type `Cell<String>`. However, that operation by itself may be perfectly harmless; perhaps the programmer knows what she is doing and only ever stores strings in the object, in which case this choice would cause an unnecessary false positive. An even more draconian choice would be to reject all uses of `dynamic` in generic class parameters, which in this example would result in a warning on line 12. Another candidate is line 14; however to report a problem at that point would require knowing that `x` and `y` are aliases, and maybe that strategy would also result in too many false positives. Instead, we choose for our analysis design to report the errors *where they may occur at runtime*, in this case line 15.

The `Cell` example involves `dynamic` in generic types; a similar issue arises for function types since the return type of function closures may be `dynamic`. In the following example, `T` is the function type `() => String`, which suggests that `f()` returns a value of type `String`.

```
18 typedef String T();
19 void main() {
20   T f = (() { return new Object(); });
21   String y = f();
22 }
```

Example 3: Dynamic return type.

The assignment in line 20 is acceptable by the standard type checker and in checked mode execution (the runtime return type of the function is `dynamic`), however, a subtype-violation runtime error occurs in line 21 because `Object` is not a subtype of `String`. In this case, we can trust, based on the type annotation of `f`, that `f` is a function but not that it returns a value that is a subtype of `String`. Further, a consequence of a design flaw in the type rule for function subtyping is that function return types cannot be trusted, even without type `dynamic` [8]. More generally, if a variable x in a Dart program has (non-`dynamic`) declared type T then we cannot always be certain that whenever the value of x is read at runtime, its type is a subtype of T, even in checked mode execution.

In the following program, field `f` in class `B2` has a different type than in the superclass `B1`, which is allowed in Dart:

```
23 class A1 {}
24 class A2 extends A1 { String s = "foo"; }
25 class B1 { A2 f = new A2(); }
26 class B2 extends B1 { A1 f = new A1(); }
27 void main() {
28   B1 x = new B2();
29   print(x.f.s);
30 }
```

Example 4: Contravariant field overriding.

The declared type `B1` of `x` suggests that `x.f.s` exists, but the program fails in line 29 at runtime because `x.f` has type

`A1`. To catch this error statically, we could require invariant field overriding (as suggested by Ernst et al. [8]) and issue a warning at line 26. However, that might be too restrictive, and not in the Dart spirit. A better option is to issue a warning at line 29 where the runtime error occurs. This can be achieved by reasoning that `x` has declared type `B1`, so it may hold a value of type `B2` and thus `x.f` may have type `A1`, which is not a subtype of `A2`. Alternatively, we can ignore the type annotation of `x` and reason entirely using the dataflow, which shows that the runtime type of `x` can only be `B2`.

As these examples demonstrate, the various ways by which type annotations can or cannot be trusted cause subtle complications for our analysis design.

5. The SafeDart Analysis

We propose three main modes of analysis that incorporate type annotations in different ways:

flow mode As a baseline, flow mode completely ignores type annotations, similar to Dart's production mode, and relies entirely on dataflow analysis (with some exceptions for native libraries, see Section 8). This analysis mode can in principle be used to statically check whether message-not-understood errors may occur in production mode execution. We introduce flow mode mainly as a starting point for explaining the following two modes.

filter mode Filter mode takes type annotations into account to model the subtype checks conducted in Dart's checked mode execution. This is accomplished by filtering the dataflow at assignments with type declarations, as hinted in Section 3. As a consequence, filter mode can in principle be used to check for message-not-understood as well as subtype-violation errors in checked mode.

modular mode Instead of filtering dataflow, modular mode uses type annotations as specifications, which generally provides better modularity properties, as suggested in Section 3.

As we shall see, it is beneficial to *combine* filter mode and modular mode to obtain better precision and more informative warning messages than running either individually.

The analysis can additionally be configured for *optimistic* treatment of type annotations, which means that the analysis blindly trusts that type annotations are correct when, for example, modeling which types may appear at a variable read operation, thereby ignoring some of the various sources of unsoundness discussed in the previous section.

Note that in the extreme case where the program being analyzed contains no type annotations, filter mode and modular mode both work in the same way as flow mode. Conversely, for a fully annotated program, modular mode analysis with the optimistic configuration resembles traditional static type checking (yet different from Dart's standard type checking, which uses implicit downcasts pervasively).

The analysis, SafeDart, is conceptually divided into two phases. First, the *inference* phase over-approximates the possible types of all expressions; second, the *check* phase emits warnings about potential message-not-understood errors and subtype-violation errors (the latter only in filter mode and modular mode) based on the inferred types. We next explain in detail how the different modes of type inference and the type checking work.

$$
\begin{array}{lrcl}
\textit{class} & C & ::= & \mathsf{class}\ N\langle\overline{U\ \mathsf{extends}\ T}\rangle\ \mathsf{extends}\ N\ \{ \\
 & & & \quad \overline{T\ f = e} \\
 & & & \quad \overline{T\ m(T\ x) \Rightarrow^F e} \\
 & & & \}\\
\textit{typedef} & D & ::= & \mathsf{typedef}\ T\ F(T) \\
\textit{expr.} & e & ::= & x \mid e.f \mid e.m \mid \mathsf{new}\ N\langle\overline{T}\rangle() \mid x = e \mid \\
 & & & \quad e.f = e \mid e(e) \mid T\ (T\ x) \Rightarrow^F e \\
\textit{type} & T & ::= & N\langle\overline{T}\rangle \mid F \mid U \mid \mathsf{dynamic}
\end{array}
$$

Figure 1: Syntax of the simplified Dart language.

5.1 A Core Language for Dart

We begin by defining a core of Dart that we use for presentation of our analysis. Figure 1 shows the syntax of the language. It uses the set of class names N, type parameter names U, field names f, method names m, function labels F, and variable names x (we let variables coincide with function parameters). We occasionally misuse notation slightly and use e.g. N as a meta-variable ranging over the set of class names rather than as the set itself. A possibly empty list of X elements is denoted by $\overline{X}$.

A program is a collection of classes C and function typedefs D. Each class contains fields and methods (collectively called properties), either explicitly defined or inherited from its superclass. Classes are parameterized by types, written in $\langle\ldots\rangle$, where each type parameter has a bound. A field f is defined with an initialization expression e and a type T, which can be an object type $N\langle\overline{T}\rangle$, a type parameter U defined for the surrounding class, a function type F (defined by a typedef), or the type dynamic. Note that F plays two roles: as names of typedefs and as labels of methods and anonymous functions. A method $T_r\ m(T_x\ x) \Rightarrow^F e$ is defined with a name m, a parameter x with type T_x, a return expression e, and a return type T_r. The notation $N.p$ refers to the property p in the class N.

Expressions e specify computations including variable and property lookup, instance creation, assignments, calls, and function expressions. A function expression of the form $T_r\ (T_x\ x) \Rightarrow^F e$ defines an anonymous function. Every method and function has a label F that is used in the flow analysis and has no effect at runtime.

Every program is assumed to be well-formed meaning that there are no unresolved references to classes, function types, type parameters, or variables and uses of generic classes have the right number of type parameters. There is one distinguished, predefined type, `Object`, which is the supertype of all types. To simplify the presentation, we also assume that every function and method has a single parameter,

$$
\begin{array}{lll}
\textit{inferred type} & \tau & ::= N{<}P{>}^{\kappa} \mid F{<}P{>}^{\kappa} \mid U^{\kappa} \\
\textit{kind} & \kappa & ::= \mathsf{C} \mid \mathsf{A} \\
\textit{type arguments} & P & ::= \overline{T} \mid ? \\
\textit{type sets} & S & ::= \{\tau\}
\end{array}
$$

Figure 2: Abstract domain.

every variable name, class name, and function name is unique, every non-variable expression is unique (alternatively, one may treat every expression as having an implicit unique label), and inherited type parameters of generic object types are never redefined. Also, we omit the null and this literals and the void return type (our implementation described in Section 8 naturally covers these language features).

We omit a formalization of the language semantics due to the limited space; see Ernst et al. [8] for a formalization of the related language Fletch. Note that, as in Fletch, a dynamic programming style can be emulated by using the type dynamic everywhere.

5.2 Abstract Domain for Types

We express our analysis using set constraints on a collection of type variables [15, 19]. For a given program component to be analyzed, we allocate a type variable for each program variable x, denoted $[\![x]\!]$, and similarly for each non-variable expression e and class field or method $N.f$ and $N.m$, respectively. For each function F, we use $F.x$ and $F.r$ to denote its parameter and its return expression, respectively.

Type variables range over sets S of *inferred types* shown in Figure 2. An inferred type τ is an object type $N{<}P{>}^{\kappa}$, a function type $F{<}P{>}^{\kappa}$, or a type parameter U^{κ}. An object type $N{<}P{>}^{\kappa}$ describes objects of class N whose type parameters P are assumed to be either a list of type arguments $\overline{T}$ or ?, the latter denoting unknown type arguments (recall that type parameters cannot always be trusted, so in some situations explained later we choose not to track them). A function type $F{<}P{>}^{\kappa}$ describes functions labeled with F defined inside a generic class with type parameters P.[1]

We distinguish between two kinds of types denoted by the κ superscript: *concrete* types C and *abstract* types A. Informally, an abstract type $\tau = T^{\mathsf{A}}$ implicitly comprises all subtypes of T, whereas a concrete type $\tau = T^{\mathsf{C}}$ describes only T itself.

This abstract domain is richer than the language of type annotations, as we allow *sets* of types (or, union types). Another difference is that we distinguish between the two kinds of inferred types, which we use in our treatment of type annotations.

6. Type Inference Constraints

The inference phase of the analysis is expressed using constraints on the type variables as shown in Figure 3. As can be seen, we use only simple subset constraints and conditional constraints ($X \subseteq Y \cap Z$ represents two constraints, $X \subseteq Y$ and $X \subseteq Z$). The least solution to the constraints can be found using standard fixpoint algorithms [15, 19, 25].

To explain the different analysis modes $m \in \{\mathsf{Flow}, \mathsf{Filter}, \mathsf{Modular}\}$ in a uniform manner, we make use of some auxiliary definitions shown in Figure 4. Intuitively, flow_m and decl_m determine what types can be inferred on the basis of dataflow and type annotations, respectively. The operator $\preccurlyeq$ is a variant of Dart's subtyping relation that takes kinds (C and A) and type parameters into account. The bind_m function used in some of the constraint rules models binding of type parameters in inferred types, as explained informally in the following subsections. We next explain each mode in turn. Due to the limited space we only explain the most interesting constraint rules.

6.1 Constraints for Flow Mode

Flow mode ignores type annotations and only propagates concrete types, similar to an Andersen-style points-to analysis [25] although inferring types instead of abstract heap locations. For $m = \mathsf{Flow}$, bind_m and flow_m are simply the identity function in their first argument and ignore the second argument, decl_m always returns the empty set of types, and $\preccurlyeq$ is the identity relation.

In Figure 3, the constraint rule [OBJ] infers the type of a new expression as a concrete object type. In this analysis mode, generic type parameters are irrelevant so we simply replace them with ?. The rules [FUN] and [METHOD] similarly infer concrete function types for function expressions and method declarations, respectively. [WRITEVAR] models the flow of types at variable assignments. Method and field reads (including method dispatch) are modeled by [READMETHOD] and [READFIELD], respectively. Field assignments are similarly handled by [FIELD] and [WRITEFIELD], and [INH1] and [INH2] model the flow of types for inherited but not overridden fields and methods. Finally, [APP] models the flow of types for the parameter and return value at call sites. Note that in this mode we conduct on-the-fly control flow analysis using the labels in the concrete function types, as in many points-to analyses.

6.2 Constraints for Filter Mode

Filter mode is different from flow mode in that inferred types are filtered based on type annotations, similar to the use of type filters in points-to analysis [25]. Revisiting Figure 4 with $m = \mathsf{Filter}$, the function $\mathit{flow}_m(S, T)$ now performs this filtering: it only admits types from the type set S that are subtypes of T, thereby modeling the effect of runtime subtype checks in Dart checked mode execution. (The relation $<:$ denotes subtyping, with special treatment of dynamic and with covariant generics, as formalized by Ernst et al. [8].) We make use of a static type resolution mechanism, type, which finds the type T that can be used in this subtype check, as explained in the following.

As an example, for a variable assignment $x = e$, we filter the dataflow using the declared type of x, that is, its

[1] A function type $F{<}P{>}$ in our notation corresponds to $(\mathit{type}(F.x, P)) \rightarrow \mathit{type}(F.r, P)$ in Ernst et al. [8].

class N ... {	
$T\ m(T_x\ x) \Rightarrow^F e$	$\{F\texttt{<?>}^{\mathsf{C}}\} \subseteq [\![N.m]\!],\ decl_m(T_x) \subseteq [\![x]\!]$ [METHOD]
$T\ f = e$	$flow_m([\![e]\!], type(N.f, ?)) \subseteq [\![N.f]\!]$ [FIELD]
}	
class M ... extends N ...	$flow_m([\![N.f]\!], type(N.f, ?)) \subseteq [\![M.f]\!]$ for each non-overridden field f [INH1]
	$[\![N.m]\!] \subseteq [\![M.m]\!]$ for each non-overridden method m [INH2]
new $N\texttt{<}\overline{T}\texttt{>}()$	$bind_m(\{N\texttt{<?>}^{\mathsf{C}}\}, \overline{T}) \subseteq [\![\text{new } N\texttt{<}\overline{T}\texttt{>}()]\!]$ [OBJ]
$T\ (T_x\ x) \Rightarrow^F e$	$\{F\texttt{<?>}^{\mathsf{C}}\} \subseteq [\![T\ (T_x\ x) \Rightarrow^F e]\!],\ decl_m(T_x) \subseteq [\![x]\!]$ [FUN]
$e.m$	$\dfrac{\tau \in [\![e]\!] \qquad N\texttt{<}P\texttt{>}^{\kappa} \preccurlyeq \tau}{bind_m([\![N.m]\!], P) \subseteq [\![e.m]\!]}$ [READMETHOD]
$e.f$	$\dfrac{\tau \in [\![e]\!] \qquad N\texttt{<}P\texttt{>}^{\kappa} \preccurlyeq \tau \qquad T = type(N.f, P)}{bind_m\big(flow_m([\![N.f]\!], T), P\big) \cup decl_m(T) \subseteq [\![e.f]\!]}$ [READFIELD]
$x = e$	$flow_m([\![e]\!], type(x, ?)) \subseteq [\![x]\!] \cap [\![x = e]\!]$ [WRITEVAR]
$e.f = e'$	$\dfrac{\tau \in [\![e]\!] \qquad N\texttt{<}P\texttt{>}^{\kappa} \preccurlyeq \tau \qquad T = type(N.f, P)}{bind_m(flow_m([\![e']\!], T), ?) \cup decl_m(T) \subseteq [\![N.f]\!],\ flow_m([\![e']\!], T) \cup decl_m(T) \subseteq [\![e.f = e']\!]}$ [WRITEFIELD]
$e(e')$	$\dfrac{F\texttt{<}P\texttt{>}^{\kappa} \in [\![e]\!] \qquad T_x = type(F.x, P) \qquad T_r = type(F.r, P)}{bind_m(flow_m([\![e']\!], T_x), ?) \subseteq [\![F.x]\!],\ bind_m\big(flow_m([\![F.r]\!], T_r), P\big) \cup decl_m(T_r) \subseteq [\![e(e')]\!]}$ [APP]

Figure 3: Core constraints for the simplified Dart language, using mode $m \in \{\mathsf{Flow}, \mathsf{Filter}, \mathsf{Modular}\}$.

type annotation or dynamic by default. We must be careful with generic types as demonstrated in Section 4. In rule [WRITEVAR], $type(x, ?)$ finds the type to be used for filtering, where the ? argument has the following meaning for type parameters that may occur in the type annotation. To keep the analysis simple, it does not track the actual types of those type parameters. (We discuss alternative designs in Section 6.4.) We therefore conservatively interpret the type parameters as dynamic when filtering. Note that it is always sound to perform less filtering in this type inference phase of the analysis, compared to what types are actually ruled out at runtime subtype checks. (When emitting type warnings in the next phase, however, we are in the opposite situation, as discussed in Section 7.) An alternative to replacing type parameters with dynamic is to use the bounds of the type parameters, but that would be unsound since they can be invalidated by the use of dynamic as type argument, so we only do that if 'optimistic' is enabled. Type parameters occurring in $[\![e]\!]$, for example if e is `new Cell<T>` inside the `Cell` class, are treated similarly.

Type parameters in object types are now relevant, unlike in flow mode. The $bind_m$ function is no longer the identity function in its first argument but now substitutes the type parameters according to its second argument. This is used, for example, in rule [OBJ]. At method read operations, $e.m$, the use of $bind_m$ in [READMETHOD] takes care of binding the type parameters in the method's type according to the type of the receiver e. For example, assume we add a method `Cell<T> h(Cell<T> p)` $\Rightarrow^{F_3}$ `p` in the `Cell` class from Example 2 and that we have $[\![\texttt{x}]\!] = \{\texttt{Cell<int>}^{\mathsf{C}}\}$ for some variable `x`. The $bind_m$ function then ensures that the expression `x.h(x)` has type $\texttt{Cell<int>}^{\mathsf{C}}$: applying rules [METHOD] and [READMETHOD] gives $bind_m([\![\texttt{Cell.h}]\!], \texttt{int}) = bind_m(\{F_3\texttt{<?>}^{\mathsf{C}}\}, \texttt{int}) = \{F_3\texttt{<int>}^{\mathsf{C}}\}$, therefore $[\![\texttt{x.h}]\!] = \{F_3\texttt{<int>}^{\mathsf{C}}\}$ and finally $[\![\texttt{x.h(x)}]\!] = \{\texttt{Cell<int>}^{\mathsf{C}}\}$ using [APP].

For a field assignment, $e.f = e'$, at rule [WRITEFIELD], we correspondingly filter the flow of types based on the declared type of $e.f$. For each object type $N\texttt{<}P\texttt{>}^{\kappa}$ in $[\![e]\!]$ ($\preccurlyeq$ is still just the identity function), $type(N.f, P)$ gives us the type of the f field in N using generic type arguments P. The use of $bind_m$ here effectively erases any type parameters that may appear in $[\![e']\!]$ by substituting them by dynamic (or their bounds, if 'optimistic').

Rule [READFIELD] is perhaps more surprising, because it involves filtering even though it is not a write operation. For an expression $e.f$, we again consider each object type $N\texttt{<}P\texttt{>}^{\kappa}$ in $[\![e]\!]$. Now, notice that we have chosen to have only one constraint variable $[\![N.f]\!]$ irrespective of whether N is generic, thereby conflating all instantiations of the generic type parameters in a context insensitive manner. However, we recover some precision by applying $flow_m$ to filter the types in $[\![N.f]\!]$ according to P. As an example, consider an expression `x.f` in a situation where $[\![\texttt{x}]\!] = \{\texttt{Cell<String>}^{\mathsf{C}}\}$

$$flow_m(S,T) = \begin{cases} S & \text{if } m = \mathsf{Flow} \vee T = \mathsf{dynamic} \vee T = \mathtt{Object} \\ \{X^\kappa \in S \mid stype(X,?) <: stype(T,?) \wedge & \\ \qquad (m = \mathsf{Filter} \vee ftype(X))\} & \\ \qquad \text{if } m = \mathsf{Filter} \vee & \\ \qquad (m = \mathsf{Modular} \wedge (ftype(T) \vee tpar(T))) & \\ \emptyset & \text{otherwise} \end{cases}$$

$$decl_m(T) = \begin{cases} \{T^\mathsf{A}\} & \text{if } m = \mathsf{Modular} \wedge (tpar(T) \vee \\ & \qquad (T = N\overline{<T>} \wedge T \neq \mathtt{Object} \wedge opt)) \\ \{N\overline{<?>}^\mathsf{A}\} & \text{if } m = \mathsf{Modular} \wedge \\ & \qquad T = N\overline{<T>} \wedge T \neq \mathtt{Object} \wedge \neg opt \\ \emptyset & \text{otherwise} \end{cases}$$

$$\tau_1 \preccurlyeq \tau_2 \text{ iff } \tau_1 = \tau_2 \ \vee \ (\tau_1 = T_1^\mathsf{C} \wedge \tau_2 = T_2^\mathsf{A} \wedge stype(T_1,?) <: stype(T_2,?))$$

Figure 4: Auxiliary definitions. (The predicate opt denotes the 'optimistic' configuration, $tpar(T)$ means that T is a type parameter, $ftype(T)$ means that T is a function type, and $stype(X,?)$ denotes the type X where all type parameters are replaced by dynamic.)

and $[\![\texttt{Cell.f}]\!] = \{\texttt{String}^\mathsf{C}, \texttt{int}^\mathsf{C}\}$ (i.e. in some instantiations of `Cell`, the `f` field holds a string and in others an integer). By applying the filtering where the type parameter `T` (i.e. the declared type of `Cell.f`) has been substituted with `String` we get $[\![\texttt{x.f}]\!] = \{\texttt{String}^\mathsf{C}\}$.

6.3 Constraints for Modular Mode

In modular mode, we attempt to use the type annotations not to filter dataflow but as an alternative to the dataflow. Consider an assignment $x = e$ where the type annotation of x is T. If T is a non-dynamic type, we let $flow_m$ yield $\emptyset$, thereby interrupting the flow of types, and in return let $decl_m$ return T^A representing all possible objects of type T including subtypes. Conversely, for $T = \mathsf{dynamic}$, $flow_m$ and $decl_m$ behave as in flow mode. There are two important exceptions to these rules, however (see Figure 4):

First, we must take into account that not all type annotations can be trusted (cf. Section 4). If T is an object type $N\overline{<T>}$ we therefore replace $\overline{T}$ by ? in this process unless 'optimistic' is enabled. Notice that with the use of type annotations in modular mode analysis, it becomes more important whether or not we choose to trust type annotations, and thereby 'optimistic' plays a bigger role than in filter mode.

Second, we choose to treat function types in the same way as in filter mode, that is, by propagating and filtering concrete function types (i.e. inferred types of form F^C). In principle, we could instead use abstract function types (of form F^A), which might be more in the spirit of modular inference, however, that would cause precision to deteriorate. If, for example, with that approach a variable `f` has declared type `F` defined by `typedef Object F(Object)`, then $[\![x]\!] = \{\texttt{F}^\mathsf{A}\}$, which comprises all functions with one argument, so resolving a call `f(...)` would trigger dataflow to all those functions, irrespective of which ones may be stored in `f`. Moreover, recall from Example 3 that function return types generally cannot be trusted. We additionally treat $T = \texttt{Object}$ in the same way as $T = \mathsf{dynamic}$, because the type $\texttt{Object}^\mathsf{A}$ subsumes all types, including functions, so using this as an inferred type would ruin the precision of the control-flow part of the analysis.

Notice that modular mode thereby involves both abstract (A) and concrete (C) types, in contrast to flow and filter mode that only use concrete types (as long as we ignore the modeling of the native library, see Section 8).

Type parameters can appear as inferred types in modular mode, unlike in the other modes. Thereby the analysis can reason modularly about, for example, type safety of the method `U id(U x) {var y = x; return y;}` (using actual Dart syntax) where `U` is a type parameter in the surrounding class. In modular mode, we get $[\![\texttt{x}]\!] = [\![\texttt{y}]\!] = \{\texttt{U}^\mathsf{A}\}$, irrespective of how `id` is called. However, now we must be careful if such inferred types flow outside the current class which defines the scope of the type parameter; in that situation we let the $bind_m$ function conservatively convert them to $\texttt{Object}^\mathsf{A}$.

The fact that abstract types implicitly comprise subtypes is captured by the definition of $\preccurlyeq$ (Figure 4). For example, $\texttt{B2}^\mathsf{C} \preccurlyeq \texttt{B1}^\mathsf{A}$ using the classes from Example 4. As a consequence, rules [ReadMethod], [ReadField], and [WriteField] essentially resolve property lookups as in class hierarchy analysis [6]. (To increase precision, $N\overline{<P>}$ in those rules implicitly ranges over only those classes that are ever instantiated in the program being analyzed.)

6.4 Discussion

Many interesting variations of the type inference mechanism exist. We now briefly discuss some of the design choices and trade-offs that remain to be explored in future work.

To restrict the analysis complexity we have chosen a context insensitive design. A more precise, but also more expensive analysis could be obtained by qualifying the type constraint variables by valuations of the generic type parameters. For the example program in Section 4, we would then have distinct type constraint variables $[\![\texttt{Cell<String>.f}]\!]$ and $[\![\texttt{Cell<dynamic>.f}]\!]$ for the `f` field of `Cell`, whereas we now only have one, $[\![\texttt{Cell.f}]\!]$. Although our simple analysis design is capable of recovering some precision at property access operations by the use of filtering, as discussed in Section 6.2, proper context sensitivity would naturally enable additional precision in other situations.

A possible alternative to the 'optimistic' option is to extend the analysis to track the flow of dynamic in generic type parameters and function types, in a way that soundness would be retained while preserving the advantages of the optimistic assumptions in typical cases.

The design of modular mode inference involves trade-offs between modularity and precision. At type annotations that cannot be fully trusted, such as, a variable declared by `Cell<String> x`, we currently opt for modularity by splitting

the dataflow and using `Cell<?>`$^{\mathsf{A}}$ as inferred type (assuming non-'optimistic'), instead of falling back to filtering on the assignments to `x`, which would generally be more precise. Another interesting design choice is whether to represent object instance creations by the types, as in our current analysis, or by allocation site, as common in dataflow analysis [5].

With modular mode inference, it would be natural to consider analyzing not only complete programs but also libraries without application code, given that library interfaces are often well annotated with types. Nevertheless, this is difficult to achieve without sacrificing soundness or precision. For example, if a public library method has a parameter with declared type `C` that is a class with a field `f`, then the application code (which is unknown to the analysis) could pass in an object whose type is some sub-class of `C` where `f` is overridden to have type `Object`. Type annotations at library interface in Dart therefore provide less information than what one may think. Studying extensions of our 'optimistic' configuration to address this challenge is an interesting opportunity for future work.

We next state three key properties about the types produced by the inference phase. Formalizing and proving these properties is beyond the scope of this paper.

Designed for soundness All three type inference modes are designed for soundness in the sense that they result in over-approximations compared to what types may actually appear in checked mode execution, assuming that 'optimistic' is disabled.

When 'optimistic' is enabled, we aim for an interesting form of *conditional* soundness: the information produced by the type inference is sound, provided that whenever a value is read at runtime from a variable or property a, then the type of the value is a subtype of the declared type of a. This turns out to be a reasonable assumption, as shown in Section 8.

Precision of the three modes Filter mode is always at least as precise as both flow mode and modular mode, with 'optimistic' being disabled. The reason is intuitively that it tracks the intersection of the type information originating from dataflow and type annotations, respectively. Also, enabling 'optimistic' can only improve precision, not degrade it.

Precision compared to the Fletch full type safety system The "full type safety" system proposed by Ernst et al. [8] is another approach to sound type checking for Dart. Its precision is incomparable to our type inference—even when restricting to Dart programs with no occurrences (explicitly or implicitly) of dynamic in type annotations, type arguments, or closure return types. As an example, the full type safety system rejects Example 4 even if removing line 29, unlike our analysis (in any mode). Conversely, the restrictions imposed by the full type safety system in some situations allow it to be more precise regarding generic type parameters.

7. The Type Checking Phase

After the type inference phase, we have a set $[\![X]\!]$ of inferred types for every constraint variable X for the program being analyzed. Based on this information, the goal of the type checking phase is now to report warnings if message-not-understood or subtype-violation errors may occur.

Message-not-understood checking At every property access operation $e.p$, emit a warning if some inferred type in $[\![e]\!]$ does not have a p property.

Type parameters in the inferred types are interpreted according to their bounds if 'optimistic' is enabled, and otherwise as `Object`$^{\mathsf{A}}$. In Dart, function calls and method calls technically involve looking up the `call` property, so in our implementation this check also detects attempts to call a non-function value or pass the wrong number of parameters.

For this message-not-understood check, filter mode is always at least as precise as flow mode and modular mode. Nevertheless, analyzing a program in both filter mode and modular mode may give more information to the programmer compared to filter mode alone. Consider, for example, a function `f(A1 x) { var y = x; return y.s; }` in a program where only `A2` objects are passed as argument to the function, `A2` is a subclass of `A1`, and the field `s` is only defined in `A2`. Using the information from both analysis modes, we can tell the programmer that the expression `y.s` is safe in the current version of the program but the code is fragile because of the misleading type annotation `A1`.

Subtype-violation checking At every assignment of some expression e to a field, variable, or function/method parameter x, emit a warning if some inferred type τ in $[\![e]\!]$ is not a subtype of the declared type of x. Occurrences of ? or `dynamic` in an inferred abstract type τ, as e.g. in `Cell<?>`$^{\mathsf{A}}$, can denote any object and are accordingly treated as `Object` in this subtype check. At method calls and field writes, type parameters in declared types are substituted according to the type of the base object. (In the full Dart language, methods may also be invoked using tear-off functions; in that case type parameters in declared types are treated as `Object`$^{\mathsf{A}}$.) Type parameters in inferred types are interpreted in the same way as for the message-not-understood checks, except at variable writes where we can take advantage of reflexivity of subtyping, as explained in the following.

Consider an assignment `x=y` where both variables `x` and `y` have declared type `T`, which is a type parameter with bound `Object` in the current class, and where the possible values of `y` may be of type `int` or `String`. Filter mode inference essentially ignores the type annotation for `y`, as explained in Section 6.2, yielding $[\![\mathtt{y}]\!] = \{\mathtt{int}^{\mathsf{C}}, \mathtt{String}^{\mathsf{C}}\}$. Now the subtype-violation check results in a warning, because the declared type of `x` is `T`. Modular mode inference, on the other hand, will conclude $[\![\mathtt{y}]\!] = \{\mathtt{T}^{\mathsf{A}}\}$. Since subtyping is reflexive, the information from modular mode thereby suffices for proving type safety of the assignment.

As this example shows, filter mode is, perhaps surprisingly, not always at least as precise as modular mode regarding subtype-violation checks. For the reasons discussed in Section 6.4, filter mode is more precise than modular mode in other situations. Thereby we can improve precision of subtype-violation checking by running *both* filter mode and modular mode inference and then emit a warning at a given assignment only if both of them fail the subtype check.

We emphasize that message-not-understood errors are more critical than subtype-violation errors, as only the former are relevant in production mode execution (recall from Section 2 that runtime subtype checking is only performed in Dart's checked mode execution, and type annotations have no effect in production mode execution). Conversely, if using the checked mode semantics, our analysis is sound with respect to message-not-understood errors independent of the precision regarding subtype-violation errors: if no message-not-understood warning is produced by our type checker at a given property access operation, then message-not-understood errors cannot occur at that operation in checked mode execution.

Examples We can demonstrate some interesting aspects of the analysis using the example programs from Section 4. Analyzing Example 2 with filter mode gives $[\![\texttt{y}]\!] = \{\texttt{Cell<}\textbf{\texttt{dynamic}}\texttt{>}^{\texttt{C}}\}$ and $[\![\texttt{y.f}]\!] = \{\textbf{\texttt{int}}^{\texttt{C}}\}$, and `int` does not have the property `substring`, so a message-not-understood warning is generated at line 15. In modular mode we have $[\![\texttt{x}]\!] = [\![\texttt{y}]\!] = \{\texttt{Cell<?>}^{\texttt{A}}\}$, $[\![\texttt{y.f}]\!] = [\![\texttt{Cell.f}]\!] = \{\textbf{\texttt{int}}^{\texttt{C}}\}$, resulting in the same warning but also a (spurious) subtype-violation warning at line 13. Notice that modular mode inference in this case automatically falls back to resolve $[\![\texttt{y.f}]\!]$ based on dataflow due to the use of dynamic.

Considering Example 3, let F_5 be the label of the anonymous function in line 20. In both filter mode and modular mode, using rules [FUN] and [WRITEVAR] gives $[\![\texttt{f}]\!] = \{F_5\texttt{<?>}\}$ and [NEW] gives $[\![F_5.r]\!] = \{\texttt{Object}^{\texttt{C}}\}$ (since $F_5.r$ is the expression `new Object()`). Rule [APP] now gives $[\![\texttt{f()}]\!] = \{\texttt{Object}^{\texttt{C}}\}$ in both modes, and therefore our type checker raises a warning on the assignment at line 21, because `Object` is not a subtype of `String`.

Analyzing Example 4, which does not involve type dynamic, with filter mode gives $[\![\texttt{x}]\!] = \{\texttt{B2}^{\texttt{C}}\}$ and $[\![\texttt{x.f}]\!] = \{\texttt{A1}^{\texttt{C}}\}$, and `A1` does not have the property `s`, again resulting in a message-not-understood warning. With modular mode we instead have $[\![\texttt{x}]\!] = \{\texttt{B1}^{\texttt{A}}\}$ and $[\![\texttt{x.f}]\!] = \{\texttt{A2}^{\texttt{A}}, \texttt{A1}^{\texttt{A}}\}$, reaching the same conclusion. Notice in this last case that looking up `f` in $\texttt{B1}^{\texttt{A}}$ involves not only the `B1` class itself but also its subclass `B2` (cf. rule [READFIELD] and the definition of $\preccurlyeq$), which would not be necessary in languages with invariant field overriding.

8. Evaluation

Our overall hypothesis is that it is possible to integrate type annotations into a flow analysis that can effectively check absence of runtime type errors in Dart programs—in a way that retains soundness even though the type annotations cannot always be trusted. More specifically, we aim to answer the following research questions through an experimental evaluation:

Benchmark	LOC excl./incl. deps.	Baseline MNU / SV
dart2js	102 718 / 104 414	60 431 / 82 571
analyzer	83 410 / 86 297	37 213 / 53 876
devcompiler	66 247 / 159 883	7 458 / 11 034
dartstyle	3 591 / 73 765	3 291 / 3 415
linter	2 236 / 77 535	938 / 1 343
petitparser	2 065 / 3 280	1 155 / 1 559
bzip2	1 105 / 2 280	665 / 1 207
coverage	847 / 3 586	446 / 667
markdown	697 / 1 846	441 / 701
crypt	161 / 1 199	118 / 197
total	263 077 / 514 805	112 156 / 155 672

Table 1: Benchmarks used for the evaluation.

1. **(precision of filter mode and modular mode)** How precise is the analysis using the different modes (when not using 'optimistic')? We focus on filter mode and modular mode; flow mode is expected to be much less effective. For the reason explained in Section 7, the precision regarding message-not-understood errors is our primary interest.
2. **(optimistic assumptions)** What is the effect of enabling 'optimistic' treatment of type annotations? This configuration is only conditionally sound (as discussed in Section 6.4), so it is also interesting to investigate whether the condition is satisfied in practice.
3. **(causes of type warnings)** What are the typical reasons for warnings? Answers to this question can be very useful to guide future work on improving precision. Experiments may tell whether to focus on, for example, context sensitivity, more precise heap modeling, or improved models of the native library. Some warnings may also indicate fragile code, as discussed in Section 7.

Although precision is our main objective, analysis time is of course also relevant.

Implementation The evaluation is conducted on a range of real-world open source Dart programs, where we exclude programs that use mirrors [2] (a mechanism for dynamic evaluations similar to reflection in Java) or heavily rely on native functions. The goal of our evaluation is not to find errors in these programs; they are presumably thoroughly tested already, so runtime type errors are unlikely to exist. Analysis precision can thus be measured by the ability to show *absence* of errors.

Table 1 shows the number of lines of code (excluding/including dependencies) for each program, together with (MNU) the total number of property access operations and (SV) the total number of assignments to variables or parameters with non-dynamic type annotation. The latter numbers can be viewed as a simple baseline for comparison: an entirely naive

Benchmark	Flow		Filter		Modular		Modular w. optimistic	
	MNU	t_{inf} / t_{chk}	MNU	t_{inf} / t_{chk}	MNU	t_{inf} / t_{chk}	MNU	t_{inf} / t_{chk}
dart2js	-	-	481 (0.8%)	198s / 13s	976 (1.62%)	433s / 15s	681 (1.3%)	279s / 13s
analyzer	-	-	67 (0.2%)	54s / 4s	306 (0.9%)	124s / 5s	134 (0.4%)	67s / 4s
devcompiler	-	-	73 (1.0%)	122s / 4s	312 (4.2%)	840s / 8s	266 (3.6%)	366s / 6s
dartstyle	-	-	10 (0.3%)	56s / 1s	183 (5.6%)	134s / 3s	54 (1.6%)	74s / 1s
linter	-	-	10 (1%)	98s / 1s	26 (2.8%)	264s / 1s	25 (2.7%)	209s / 3s
petitparser	249 (21.5%)	53s / 0s	168 (14.6%)	12s / 0s	178 (15.4%)	51s / 0s	171 (14.8%)	43s / 1s
bzip2	33 (5%)	38s / 0s	2 (0.3%)	10s / 0s	31 (4.6%)	37s / 0s	2 (0.3%)	29s / 0s
coverage	36 (8%)	49s / 0s	7 (1.6%)	12s / 0s	24 (5.4%)	45s / 0s	16 (3.6%)	36s / 0s
markdown	44 (10%)	36s / 0s	4 (0.9%)	10s / 0s	28 (6.4%)	35s / 0s	11 (2.5%)	28s / 0s
crypt	0 (0%)	5s / 0s	0 (0%)	5s / 0s	1 (0%)	9s / 0s	0 (0%)	8s / 0s
total	-	-	819 (0.7%)	577s / 23s	2 036 (1.8%)	1972s / 32s	1 349 (1.2%)	1139s / 28s

Table 2: Experimental results on checking for potential message-not-understood errors using different analysis modes.

type checker would report a message-not-understood warning at each of the property access operations and a subtype-violation warning at each of the assignments.

Our implementation, benchmarks, and all experimental data are available online.[2] To solve type inference constraints we use a basic fixpoint algorithm with difference propagation [25]. The set of inferred types for a type variable may become large if precision is lost during analysis. To improve scalability we therefore widen sets containing more than k inferred types to $\{Object^{\mathsf{A}}\}$, where k is some threshold (using $k = 100$ in the experiments).

The step from the simplified language that we use for presenting the analysis in Section 5 to the full Dart language involves several interesting issues, some of which are crucial for making the algorithms practical.

Programs are represented using SSA-form, which regains some degree of flow sensitivity [14]. In the simplified language we abstracted away from control structures, but programs written in dynamically typed languages often use type tests in branch conditions. For this reason, we recognize common patterns, such as, the `is` operator, and use them for filtering types analogous to the type promotion mechanism in Dart's standard type checker [7].

It is particularly important to model the native library, which is used in all Dart programs. To this end, we exploit the type annotations in the library API and treat them as in optimistic modular mode, even in flow mode and filter mode. Native container classes (`List`, `Set`, and `Map`) are treated specially using allocation-site abstraction and extra type constraint variables representing the container contents.

To increase confidence in our implementation, we have conducted a large number of soundness tests where we execute the test suites that accompany some benchmark programs and check that the types observed at runtime are included in those inferred by our analysis.

8.1 Precision of Filter Mode and Modular Mode

We first measure the ability of the analysis to check absence of message-not-understood errors. The MNU columns below Filter and Modular in Table 2 show the number of warnings in filter mode and modular mode, respectively (without 'optimistic'), and in percentage relative to the baseline from Table 1. In total, filter mode gives only 819 warnings for the 112 156 property access operations that exist in the programs; that is, it is able to show type safety of 99.3% of these operations. Modular mode reaches 98.2% (2 036 warnings) in comparison, however, it should be taken into account that modular mode is designed not only to report type errors that can possibly occur in some execution, but also to report instances of fragile code where type annotations may be misleading, as discussed in Section 7.

In comparison, the ordinary Dart type checker gives *no* guarantees, so showing type safety of 99.3% of the property access operations is a notable achievement.

The corresponding numbers for subtype-violation warnings are 5.1% for filter mode and 10.1% for modular mode. (We omit the details due to the limited space.) As suggested in Section 7 it may be beneficial for subtype-violation checking to combine the two modes; this reduces the number of warnings to 4.2%, thus demonstrating that this trick does lead to improved precision in practice. Our analysis can consequently be used for eliminating 95.8% of the runtime checks in checked mode execution. We have not measured the effect on execution time, but it is possible that checked mode thereby becomes essentially as fast as production mode.

Flow mode ignores type annotations and has mainly been used as a starting point for explaining the other modes. Not surprisingly, context insensitivity and other sources of abstraction cause an explosion in the sizes of the inferred type sets with this inference mode, resulting in significantly worse precision and efficiency. In Table 2, '-' indicates a 10 minutes cut-off. One way to interpret this result is that incorporating type annotations adds significant value compared to merely tracking dataflow.

Table 2 also shows the time for inference (t_{inf}) and message-not-understood checking (t_{chk}), running on a 3.4 GHz i7-3770 Linux machine with 16 GB RAM. Although speed has not been an objective for this prototype implementation, we see that both filter and modular mode analysis are already

[2] `http://www.brics.dk/safedart/`

fast enough for practical use during Dart software development. Nevertheless, we believe more efficient representations of inferred types in the implementation can improve the analysis time, which we will investigate in future work.

8.2 Optimistic Assumptions

The 'optimistic' configuration allows the inference phase to trust type annotations, which is particularly relevant in modular mode as discussed in Section 6.3. As evident from Table 2, this configuration has a significant effect on precision (and it also improves analysis time). The number of message-not-understood warnings is reduced from 1.8% to 1.2%, and for subtype-violation warnings from 10.1% to 5.4%.

To investigate whether the 'optimistic' assumptions are reasonable in practice, we have conducted an extra experiment using the test suites that we also exploit for soundness testing as mentioned above. In this experiment, the benchmarks are executed extensively in a special "super-checked mode" where we inject a type cast at every variable/property read, checking that the types observed at runtime match the declared types. For instance, running Example 2 in this way triggers a cast error when reading the `f` field in line 15. Not a single violation is detected when running the test suites, which gives confidence that the 'optimistic' assumptions are indeed reasonable and do not restrict Dart programmers in using the dynamic features of the language.

8.3 Typical Causes of Type Warnings

We have manually investigated a random selection of the reported type warnings. Classifying the precise causes of such warnings is naturally difficult to do objectively, but we have observed some interesting patterns that may provide opportunities for future improvements of the analysis.

- Generic classes are a typical source of imprecision. While the type inference is able to track type parameters to some extent (cf. Section 5), we find that complex generic patterns used throughout the benchmarks by way of, for example, Dart's async and collection library, are affected by the lack of context sensitivity.
- Another important fraction of warnings relates to insufficient modelling of native libraries. Falling back to the annotated types turns out not always to be the best solution, since native libraries often provide very limited information about the outflowing types, or are very polymorphic by their nature (e.g., Dart's html library).
- The relatively high number of warnings for *petitparser* appears to be caused by a heavy use of higher-order functions.
- Type tests appear in various forms, beyond the patterns our implementation currently recognizes. Beside simple tests on local variables, we also found tests involving fields and global variables, which are not covered by SSA-form and require further reasoning.
- We also observed a number of subtle invariants on types, which cannot easily be captured by a static analysis. For example, the *dart2js* benchmark contains many occurrences of boolean fields used as type indicators. As another example, Dart's standard command line parser returns a dictionary whose string keys indicate the value type.
- As expected, modular mode analysis gives a number of warnings that point out fragile code involving misleading type annotations.

9. Related Work

The idea of applying static analysis to check for potential type errors in dynamically typed languages has a long history [1, 4, 9, 10, 16, 20, 22]. Common to most of these techniques is that they do not incorporate type annotations. A notable exception is the algorithm by Siek and Vachharajani [22], however, being based on unification, it does not work in presence of subtyping. The more recent algorithm by Rastogi et al. [20] supports object-oriented programming, but neither generics nor the kinds of unsoundness that exist in Dart's type system. The Flow static type checker uses a mix of optional type annotations and type inference to find type errors in JavaScript programs, but without any soundness guarantees [9]. The Mypy project [27] applies optional typing to Python programs for annotating and inferring types, which can be used for type checking. While similar issues arise due to Python's dynamic nature, Mypy's type checking interestingly aligns itself far more with traditional static typing compared to this paper, ruling out many dynamic programming idioms.

Related ideas have been presented for StaDyn, a variant of C# with optional typing but without higher-order functions [18], and for an extension of Dylan with function types and parametric polymorphism [17].

Gradual typing has become a popular approach to integrate type annotations into dynamically typed programming. The traditional view on gradual typing is that it allows program code with type annotations to be type checked statically, while postponing the remaining checks to runtime [21]. A recent survey by Siek et al. [24] discusses numerous variations of gradual typing that have been proposed. The lack of soundness in Dart's type system makes an unconventional design, and it takes a pragmatic view on blame tracking [23], but the language is gaining momentum in industrial software development unlike e.g. Typed Racket [26].

The *dart2js* and *dartanalyzer* tools by Google can analyze Dart programs to check for various kinds of errors, going beyond the static type checking prescribed by the language standard [12, 13]. The *dart2js* tool has an option *trust-type-annotations*, which appears to be related to our filter mode, however, not much documentation exists for these analyzers. Another interesting recent initiative by the Google Dart team is *strong mode*, which defines a subset of Dart with "a stricter, sounder type system" and a limited form of type inference [11].

Our use of type annotations as filters is inspired by previous work on alias analysis and call graph construction algorithms. The notions of concrete and abstract types in Section 5 correspond to the point and cone types, respectively, used by Sridharan et al. [25].

10. Conclusion

Optional typing has traditionally been viewed as a compromise between static and dynamic type checking where program code with type annotations is checked statically and the rest is checked dynamically. We have demonstrated that it is possible for a realistic programming language with optional typing to incorporate the type annotations into a flow analysis to provide static type checking for program code that is not fully annotated. The various reasons by which type annotations can or cannot be trusted in Dart programs lead to interesting challenges to the design of such an analysis. We have proposed two main techniques: filter and modular mode, with different strengths and weaknesses.

Our experimental results show that this is a viable approach. For example, in filter mode the analysis is able to show for 99.3% of the property access operations in the benchmark programs that message-not-understood errors cannot occur at runtime. This is a notable result, since Dart's standard type checker is unsound by design and does not provide *any* guarantees even when it produces no warnings. Similarly, the analysis makes it possible to eliminate 95.8% of the runtime subtype checks in checked mode execution.

The number of type warnings is reduced significantly when enabling the optimistic assumptions in modular mode, and these assumptions appear to be reasonable in practice. This indicates that it may be beneficial to extend the analysis to track the flow of `dynamic` in generic type parameters and function types, such that soundness can be retained while preserving the advantages of the optimistic assumptions. Other opportunities for future work include exploring the design choices and trade-offs suggested in Section 6.4, and applying the analysis for safely eliminating costly runtime type checks in compilation from Dart to low-level languages. We believe our results also provide insight into the use of the dynamic language features in Dart, which may guide the further development of the language.

Acknowledgments 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] O. Agesen. The cartesian product algorithm: Simple and precise type inference of parametric polymorphism. In *Object-Oriented Programming, 9th European Conference (ECOOP)*, 1995.

[2] G. Bracha and D. Ungar. Mirrors: design principles for meta-level facilities of object-oriented programming languages. In *Proc. ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2004.

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

[4] R. Cartwright and M. Fagan. Soft typing. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 1991.

[5] D. R. Chase, M. Wegman, and F. K. Zadeck. Analysis of pointers and structures. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 1990.

[6] J. Dean, D. Grove, and C. Chambers. Optimization of object-oriented programs using static class hierarchy analysis. In *Proc. Object-Oriented Programming, 9th European Conference (ECOOP)*, 1995.

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

[8] 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 Dynamic Languages Symposium (DLS), 2015.

[9] Facebook Inc. Flow – a static type checker for JavaScript, 2016. `http://flowtype.org/`.

[10] C. Flanagan, M. Flatt, S. Krishnamurthi, S. Weirich, and M. Felleisen. Catching bugs in the web of program invariants. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 1996.

[11] Google Inc. Strong mode, 2015. `https://github.com/dart-lang/dev_compiler/blob/master/STRONG_MODE.md`.

[12] Google Inc. dart2js, 2016. `https://www.dartlang.org/tools/dart2js/`.

[13] Google Inc. dartanalyzer, 2016. `https://github.com/dart-lang/analyzer_cli`.

[14] R. Hasti and S. Horwitz. Using static single assignment form to improve flow-insensitive pointer analysis. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 1998.

[15] N. Heintze. Set-based analysis of ML programs. In *LISP and Functional Programming*, pages 306–317, 1994.

[16] S. H. Jensen, A. Møller, and P. Thiemann. Type analysis for JavaScript. In *Proc. 16th International Static Analysis Symposium (SAS)*, 2009.

[17] H. Mehnert. Extending Dylan's type system for better type inference and error detection. In *International Lisp Conference (ILC)*, 2010.

[18] F. Ortin. Type inference to optimize a hybrid statically and dynamically typed language. *Comput. J.*, 54(11):1901–1924, 2011.

[19] J. Palsberg and M. I. Schwartzbach. Safety analysis versus type inference. *Inf. Comput.*, 118(1), 1995.

[20] A. Rastogi, A. Chaudhuri, and B. Hosmer. The ins and outs of gradual type inference. In *Proc. 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 2012.

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

[22] J. G. Siek and M. Vachharajani. Gradual typing with unification-based inference. In *Proc. Symposium on Dynamic Languages (DLS)*, 2008.

[23] J. G. Siek, P. Thiemann, and P. Wadler. Blame and coercion: together again for the first time. In *Proc. 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2015.

[24] 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.

[25] M. Sridharan, S. Chandra, J. Dolby, S. J. Fink, and E. Yahav. Alias analysis for object-oriented programs. In *Aliasing in Object-Oriented Programming. Types, Analysis and Verification*, volume 7850 of *LNCS*, pages 196–232. Springer, 2013.

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

[27] The Mypy Project. mypy – optional static typing for Python, 2016. `https://mypy-lang.org/`.